一級継続のスタックベース処理系向け実装手法

鵜川始陽

一級継続のスタックベース処理系向け実装手法

鵜川 始陽

概 要

本論文は，関数フレームをスタック上に生成するプログラミング言語処理系に，一級継続 (first-class continuation) を実装する手法についての研究成果を取りまとめたものである．本研究では，実行効率のよい実装方法であることと，可能な限り継続の意味を変えずに実装することを目標とした．以下に概要を示す．

本研究では，他言語との相互呼出し機能を持つ処理系のための，一級継続および関連する制御機能の実装手法を提案する．また，関数フレームをスタック上に生成する一般的な処理系のための，効率のよい一級継続の実装手法である遅延スタックコピー法 (lazy stack copying) を提案する．さらに，遅延スタックコピー法のさらなる効率化や，遅延スタックコピー法のオーバヘッドを軽減する手法についても提案する．

他言語との相互呼出し機能を持つ処理系上の実装については，Java 言語と相互呼出し可能な Scheme 言語の処理系「ぶぶ」を対象とした．ぶぶは Java で記述された Scheme インタプリタである．Scheme の関数フレームは，Java の配列を使って実装された Scheme 用のスタックに生成する．ぶぶでは，Scheme の関数から Java のメソッドを呼び出し，さらに，Java のメソッドから Scheme の関数を呼び出すことが可能になっている．しかし，Java プログラムの継続は，ポータブルな方法では一級オブジェクト (first-class object) として扱うことができない．そのため，ぶぶでは無限の寿命を持つ完全な一級継続を扱うことができない．本研究では，Java 言語のポータビリティを維持したまま実装可能で，かつ，可能な限り多くの場面で利用できる一級継続を提案する．この継続は，継続キャプチャ後，Java のメソッドにリターンするまでの間は完全な継続として振る舞う．リターン後や，キャプチャしたスレッド以外のスレッドでは，キャプチャした時点から Java のメソッドにリターンするまでの計算を表わす部分継続 (partial continuation) として振る舞う．さらに，Java の例外処理機構とシームレスな例外処理機構と，継続を使った大域的な制御の移動の際に，前処理と後処理を行う機能の実装方法も提案する．

遅延スタックコピー法は，関数フレームをスタック上に割当てる処理系のための，効率のよい一級継続の実装手法である．関数フレームをスタック上に生成する処理系では，関数からリターンすると，関数フレームが解放されてしまう．そこで，従来の一級継続の実装では，継続がキャプチャされたときに関数フレームのコピーを生成し，将来の継続の呼出しに備えていた．しかし，関数フレームが解放されるより前にしか呼び出されない継続では，このコピー操作は必要ない．提案手法では，関数フレームが解放される直前までコピー操作を遅延する．もし，関

数フレームをコピーする前に継続が必要なくなったことが分かれば，コピー操作はしない．これにより，関数フレームのコピーに要する時間や，コピー先の記憶領域を節約することができる．

さらに，遅延スタックコピー法を拡張して，一つの関数フレームのコピーを複数の継続で共有することで時間や記憶領域を節約する手法を提案する．関数フレームのコピーを共有する一級継続の実装戦略は，既にいくつか提案されている．しかし，これらの実装戦略は特殊な処理系の構造を必要とするため，関数フレームをスタック上に生成する一般的な処理系に導入するには，処理系の構造を大きく変更しなければならなかった．提案手法を使えば，一般的な処理系でも，局所的な変更で関数フレームのコピーが共有できる．

本研究では，あらゆるヒープへの書き込みを見張るエスケープバリア (escape barrier) を使い遅延スタックコピー法を実現した．エスケープバリアはオブジェクトの生成も見張るため，一級継続を使わないプログラムにも相当のオーバヘッドがかかる恐れがある．そこで，プログラムを実行前に解析し，バリアの少ないプログラムへと変換する手法についても提案する．

目 次

図目次

第1章　はじめに

継続（continuation）とは，計算のある時点での残りの計算を表したものである．継続は形式的にプログラムの意味を考える際に，プログラムの実行順序を表すために非常に重要な役割を果たす．一方，プログラミング言語に，継続を一級オブジェクト（first-class object）として扱う機能を持たせることがある．プログラマは一級継続（first-class continuation）を使うことで，マルチタスクや例外処理など，ユーザレベルで実現することができる．Scheme言語は，一級継続を言語仕様に持つプログラミング言語である．Scheme言語では，一級継続以外の複雑な制御機能は持たない，シンプルな言語仕様であるが，他のプログラミング言語で可能な制御の流れ（control flow）を実現することができる．さらに，一級継続を持たないプログラミング言語では，その言語に制御機能として備わっていない制御の流れを実現するには困難な場合が多い．これに対して，一級継続を持つプログラミング言語では，一級継続を使うことで，どのような制御の流れでも実現することができる．このように，継続を一級オブジェクトとして，プログラムから直接扱えるようにすることは，記述性の面で非常に有益である．

しかし，一般にプログラミング言語処理系に一級継続を実装することは難しく，処理系を実装する言語の制約条件などによっては，不可能な場合もある．また，実用的なプログラミング言語では，実行効率がよいことが必要不可欠であるが，効率のよい一級継続を実装することはさらに困難である．以上のような理由から，効率のよい一級継続の実装手法を開発することが重要である．

この章では，まず本研究の背景と動機を述べる．次に，本論文の構成と各章の概要を示し，最後に，本研究の成果をまとめる．

1.1　背景と動機

ほとんどのプログラミング言語では，プログラムの実行の順序は構文により決まっており，制御の流れを直接プログラムから操作することはできない．このようなプログラミング言語では，一般的に有用な制御の流れのパターンを実現した専用の言語機能を備え，これらを使うことで，間接的にプログラムから制御の流れを操作できるようにしている．例えば，近代的なプログラミング言語として有

名なJava言語[1]では，例外処理（exception handling）機構や言語レベルのマルチスレッドを提供している．プログラマはこれらを使い，実行中の計算を打ち切って，特定の位置に制御を移したり，複数の計算を並行して進めたりすることができる．このようなプログラミング言語では，言語の備える制御機能はその言語が使用される場面や，言語が設計された当時に発明されていた制御パターン，計算機の能力などにより選択されている．

一方，制御（control）を抽象化した継続という概念がある．継続とは，計算のある時点での残りの計算を表す概念である．継続の概念を使うと，あらゆるプログラムの実行の順序を表現することができる．したがって，この継続をプログラムから直接操作すれば，あらゆる制御の流れを実現することができる．継続が一級オブジェクトであるプログラミング言語には，Scheme言語[2, 3]やSmalltalk-80言語[4]，また，ML言語の一実装であるStandard ML/New Jersey[5]などがある．Scheme言語は，シンプルな言語仕様を目指して設計された言語であり，関数呼出しと一級継続以外の複雑な制御機能を言語仕様に含んでいない．それでも，プログラマは関数呼出しと一級継続を利用して，あらゆる制御の流れを比較的簡潔に記述することができる．

この二つの制御機能実現の方針には，それぞれ長所と短所がある．有用な制御の流れのパターンを個別に言語機能として取り込んだプログラミング言語は，そのパターンに合致するプログラムを簡潔に記述できる利点がある．また，その言語の処理系を実装する際，採用した制御機能の性能に最適化できる可能性がある．しかしこの方針では，多くの言語機能を採用すると，言語仕様が膨らみプログラマが言語を修得するのが難しくなる．逆に，採用する制御機能を減らすと，言語が採用していない制御の流れが必要となるプログラムを記述する際に問題が起こる．したがって，用途の決まっているプログラミング言語には適していると言えるが，多目的のプログラミング言語では記述性に問題がある場合がある．

逆に，一級継続を持つプログラミング言語では，他の制御機能を持たなくても十分な記述能力を持つため，言語仕様がシンプルになる．また，どのような制御の流れでもユーザレベルで実現することができる．そのため，多目的のプログラミング言語では記述性の点で，一級継続を持たないプログラミング言語よりも有利である．この点は，Scheme言語のような一般のプログラミング言語でも当てはまるが，汎用の中間言語ではさらに重要になる．例えばMicrosoft intermediate language（MSIL）[6]は，Microsoft社が提唱するMicrosoft .NET環境で使用される中間言語である．様々なプログラミング言語のコンパイラが，それぞれの言語をMSILにコンパイルすることを想定している．また，Java言語の中間言語であるJavaバイトコード[7]も同様の性格を持つ中間言語である．Javaバイトコードは，本来はJava言語用の中間言語であったが，近年，Java言語以外の言語からも

Javaバイトコードにコンパイルし，共通のJava仮想マシンで実行させようという研究が行われている[8]．MSILやJavaバイトコードにコンパイルされたプログラムは，ソースプログラムがどのような言語で記述されていても，共通の仮想マシンにより実行される．しかし，汎用の中間言語の場合どのような言語から変換されるかを設計時に想定することができない．ソース言語から中間言語にコンパイルする際，ソース言語の制御機能が中間言語で直接表現可能であれば，効率のよい中間言語のプログラムに変換できる．直接表現できなくても，CPS変換により関数クロージャ（closure）を使って継続を表現し，任意の制御の流れを実現するか，または，中間言語でインタプリタを記述することで，共通の仮想マシン上でソースプログラムを実行することはできる．しかし，これらの場合，一般に実行効率は悪くなる．そのため，このような中間言語では一級継続を持ち，あらゆる制御の流れを直接表現できることが望ましい．

制御機能として一級継続のみを持つプログラミング言語の短所も，有用な制御の流れのパターンを個別に言語機能として取り込んだプログラミング言語の長所に対応する．一級継続のみを持つプログラミング言語では，多く使われる制御の流れのパターンでも，一級継続を使ってユーザレベルで実装しなければならないため，そのパターンを言語機能として持っている言語と比べると，記述性は劣る．また，そのパターンについての性能は劣る可能性がある．しかしこの問題は，マクロやライブラリを用意することで記述性を高めたり，多く使われると思われるパターンは専用の制御機能を用意し，効率を良くしたりすることで解決できる．もう一つの問題は，効率のよい一級継続の実装が難しいことである．例えば，既存の一級継続の実装手法であるGC戦略（garbage collection strategy）[5]は，一級継続を使ったプログラムは効率良く実行できるが，一級継続を使わないプログラムへのオーバヘッドが大きい．スタック戦略（stack strategy）[9]では，一級継続を使ったプログラムの効率が非常に悪い．また，インクリメンタル・スタック/ヒープ戦略（incremental stack/heap strategy）[10, 11]は，性能は良いが，実装が難しい．また，インクリメンタル・スタック/ヒープ戦略は，スタックの操作が特殊で，一般的なスタックベースの処理系への実装には大きな変更が必要となる．なお，これらの実装戦略の詳細は2.4節で述べる．さらに，他の言語機能と組み合わせると一級継続の実現自体が難しくなる場合や，継続の意味が曖昧になる場合もある．例えば，並列プログラムにおける継続の意味は自明ではない．Producer-consumer型の計算のように，二つのスレッドが独立した計算を行っていると考えれば，各スレッドでの継続はそのスレッドの終わりまでの計算である．しかし，fork-join型の計算のように，あるスレッドが行っている計算が全体の仕事のうち一部である場合には，そのスレッドの終わりまでの計算というのは，全体の継続のうち一部でしかない．別の例で，高級言語から低級言語の関数を呼び出す，ネイティブプ

ログラミングインタフェース（native programming interface）を持つ場合，低級言語の継続を一級オブジェクトとして扱うことができるとは限らない．

これらの問題を解決するため，本研究では，マルチスレッドとネイティブプログラミングインタフェースを持つ Java 上の Scheme 処理系での，一級継続の実装方法について提案する．マルチスレッドと一級継続の組み合せについては，既に文献 [12] などの研究がある．本研究では，ネイティブプログラミングインタフェースとの組みも合せを中心にした．また，スタックベースの一般的な処理系で容易に実装することができる，効率のよい一級継続の実装手法である遅延スタックコピー法（lazy stack copying）と，その拡張のスタックコピー共有法（stack copy sharing）を提案する．

1.2 論文の構成

本論文では，プログラミング言語処理系に一級継続を実装する手法を提案する．

まず，第 2 章で一級継続について説明する．ここでは，一級継続を使ったプログラミングや，既存の一級継続の実装手法に触れる．

次に，第 3 章で，ネイティブプログラミングインタフェースとマルチスレッド機能を持つ処理系での一級継続の実装方法を提案する．ここでは，ネイティブプログラミングインタフェースとマルチスレッド機能を持つ処理系として，Java 言語で記述された Scheme 言語処理系「ぶぶ」を使う．ぶぶでは，Java 言語のオブジェクトを一級オブジェクトとして扱うことができる．ネイティブプログラミングインタフェースとしては，メソッドの呼出しや，Scheme 言語で記述したメソッドによるメソッドのオーバライド（override）が可能である．また，Java 言語のスレッド機能も利用できる．このような処理系で，実用的な一級継続を設計し，その実装方法を示す．

第 4 章～第 6 章では，関数フレームをスタック上に生成する処理系で，効率のよい一級継続を実装する手法を提案する．関数フレームをスタック上に生成する処理系では，一級継続に無限の寿命を与えるために，関数フレームをスタックの外にコピーしなければならない．そのため，従来の多くの一級継続を実装した処理系では，継続を表すオブジェクトが作られると，直ちに関数フレームをヒープにコピーしていた．しかし，実際には，継続を表すオブジェクトが関数フレームのコピーを持っていなくてもプログラムを正しく実行できる場合がある．結果的に一度も呼び出されなかった継続や，関数フレームがスタック上に残っている間にしか呼び出されなかった継続がそれである．第 4 章では，この点に着目し，関数フレームのコピーを遅延させることにより，一級継続を使ったプログラムの実行効率を高め手法である，**遅延スタックコピー法**（lazy stack copying）を提案する．

第5章では，遅延スタックコピー法を改良し，より実行効率を高める手法として，**スタックコピー共有法**（stack copy sharing）を提案する．この手法は，一度生成した関数フレームのコピーを何回も再利用することで，一級継続を使ったプログラムの実行効率を高める．関数フレームのコピーを再利用することで，実行効率が良くなることは既に知られており，そのような実装戦略も提案されている[9, 10, 11]．しかし，ソースコードレベルでのポータビリティの高いプログラミング言語では，制御スタックやレジスタを，プログラムから直接参照することができない．そのため，このようなポータビリティの高い言語へのコンパイラや，ポータビリティの高い言語で記述されたインタプリタでは，既存の実装戦略が使えなかった．スタックコピー共有法は，このような処理系でも関数フレームの共有が可能になり，一級継続を使ったプログラムの実行効率が良くなる．

第4章では，エスケープバリア（escape barrier）と呼ぶバリア（barrier）を使い，遅延スタックコピー法を実現する．エスケープバリアは，オブジェクトの生成時に，生成されるオブジェクトのスロットの初期値となる値をチェックする．もし，その値が継続オブジェクト（continuation object）への参照であれば，継続オブジェクトにスタックの外から参照されたことを表す印を付ける．オブジェクトの生成が軽い処理系では，エスケープバリアのオーバヘッドが相対的に大きくなる恐れがある．そこで第6章では，プログラムを実行前にエスケープバリアの少ないプログラムに変換する手法を提案する．ここでは，データフローを調べ，継続オブジェクトへの参照とはなり得ない値や，既にチェックされている値をチェックするバリアを除去する．

最後に，第7章で本論文をまとめる．

1.3 研究の成果

本研究では，ネイティブプログラミングインタフェースとマルチスレッドを持つ処理系への一級継続の実装手法と，遅延スタックコピー法やその改良法を提案した．ネイティブプログラミングインタフェースとマルチスレッドを持つ処理系では，完全な一級継続が実装できない場合が多いが，本研究で提案する手法を使えば，多くの場面で有効に利用できる一級継続を実現することができる．また，遅延スタックコピー法は関数フレームをスタックに生成する処理系に効率の良い一級継続を実装する手法である．本研究で示した改良と併用することで，様々な処理系において効率の良い一級継続を実装することができる．また，そのオーバヘッドも小さく抑えることができる．

第2章　継続

本論文では，一級継続の実装手法を扱う．そこで，この章では継続について説明する．まず，継続や一級継続を紹介し，本論文で用いる用語を定義する．本論文では，一級継続を持つプログラミング言語としてScheme言語を使う．そこで，次に，Scheme言語を簡単に紹介する．さらに，一級継続を使ったプログラミングの例を紹介し，一級継続の記述性の高さを示す．最後に，既存の一級継続の実装戦略の中から代表的な実装戦略として，GC戦略，スタック戦略，インクリメンタル・スタック/ヒープ戦略の三つの戦略を紹介する．

2.1　一級継続

計算のある時点での，残りの計算全体を**継続**（continuation）と呼ぶ．プログラムの実行中は常に，実行中の式の結果を待っている継続が一つ存在する．これを，**現時点の継続**（current continuation）と呼ぶ．式の実行が終わると，現時点の継続が起動され，次の式の実行が始まる．式を実行するために別の式の実行が必要な場合,「実行中の式の残りを計算した後，現時点の継続を起動する」という，より大きな計算を持つ継続が現時点の継続となり，目的の式の実行が始まる．

本論文では，次のように継続の半順序を定義する：

> 継続 c_1 と c_2 について，継続 c_1 が,「ある計算をした後継続 c_2 を起動する」という計算を持つ場合，c_1 は c_2 を**含む継続**または，c_2 より**大きい継続**と呼び，$c_1 \sqsubseteq c_2$ と書く．実際には，プログラムの実行はサボテン状の関数呼出しにより行われるため，任意の継続 c_1 と c_2 について，c_1 と c_2 の両方に含まれる継続 c_3 と，その最大元 c_3'，つまり，$c_1 \sqsubseteq c_3'$ かつ，$c_2 \sqsubseteq c_3'$ かつ，任意の c_3 について，$c_3' \sqsubseteq c_3$ であるような c_3 が存在する．このような c_3 を，c_1 と c_2 が**共通に含む継続**と呼ぶ．また，c_3' を，**最大の共通に含む継続**と呼び，$c_1 \sqcup c_2$ と書く．さらに，現時点の継続に含まれる継続を**アクティブな継続**と呼ぶ．

ほとんどのプログラミング言語では，式の実行の開始と継続の呼出しの連鎖は暗黙に行われて，プログラマから特別に意識されることはない．代わりに，`exit`

や `return` や `goto`，さらに例外処理機構などの専用の言語機能を用意し，これを使って間接的に現時点の継続以外の継続を起動することを可能としている．

一方，Scheme 言語などは**一級継続**（first-class continuation）を持つ．つまり，任意の関数呼出しの際に，現時点の継続を表す一級オブジェクトを生成し（継続の**キャプチャ**（capture）と呼ぶ），プログラムから直接操作できる．キャプチャされた継続を表す一級オブジェクトを，**継続オブジェクト**（continuation object）と呼ぶ．継続オブジェクトは，任意の時点で**呼び出す**ことができる．継続オブジェクトが呼び出されると現時点の継続は捨てられ，代わりに呼び出された継続オブジェクトが持つ継続の計算が始まる．つまり，呼び出された継続オブジェクトの持つ継続（これを単に呼び出された継続と呼ぶことにする）がキャプチャされた時点に制御が移動する．一級継続を使うと，先に挙げた `exit` などの他，マルチタスクなど，あらゆる制御の流れをプログラムレベルで実現することができる．2.3 節では，継続を使って例外処理とマルチタスクを実現する例を紹介する．

一級継続を持つプログラミング言語は，Scheme 言語の他，Smalltalk-80 言語 [4] や ML 言語の一実装である Standard ML/New Jersey[5] などがある．また，Java 言語に一級継続を追加する研究 [8] もある．C 言語でもライブラリ関数の `setjmp` と `longjmp` を使うことで，継続のキャプチャと呼出しができる．しかし，`setjmp` でキャプチャした継続は，`setjmp` を呼び出した関数からリターンするまでの間しか使うことはできない．そのため，C 言語の `setjmp` と `longjmp` は，それだけでは，あらゆる制御の流れを表現することはできない．本論文では，利用に制限のない，無限の寿命を持つ継続を対象とする．

2.2 Scheme 言語

本論文では，一級継続を持つ言語の例として Scheme 言語を用いる．そこでこの節では，Scheme 言語について簡単に説明する．詳細については，Scheme 言語の仕様書 [2, 3] を参照されたい．Scheme 言語は，関数型のプログラミング言語で，次のような特徴を持つ．

- 最小限のプリミティブを備えたシンプルな仕様である．
- **真正に末尾再帰的**（properly tail recursive）である．
- 関数や関数クロージャが一級オブジェクトである．
- 継続を一級オブジェクトとして扱うことができる．
- 自動的に不要なオブジェクトは解放される．これは，一般にはごみ集めにより行われる．

- プリント式や代入式などの副作用を持つ式や，逐次実行など，純粋に関数型でない機能も持つ．

ここで，真正に末尾再帰的とは，無限個の**アクティブな末尾呼出し**（tail call）が可能であるということである．関数呼出しがアクティブであるとは，呼び出された関数か，そこから呼び出された関数に制御があるということである．Scheme 言語では，関数からリターンした後でも関数内でキャプチャした継続を呼び出すことで，再び呼出しがアクティブになる可能性がある．次に，末尾呼出しとは直感的には関数の最後で，それ以上関数の計算が残っていない位置（これを**末尾位置**（tail context）と呼ぶ）での関数呼出しのことである．Scheme 言語の末尾位置は，正確には次のように定義されている [3].

- ラムダ式の本体の最後の式は，そのラムダ式の表す関数の末尾位置に出現している．
- `if` 式など，いくつかの式が末尾位置に表れている時，式ごとに決められた位置は末尾位置となる．ここでは，`if` 文のみ紹介する．`if` 文では，次の構文定義の ⟨tail expression⟩ で示された位置の式が末尾位置にある．

  (`if` ⟨expression⟩ ⟨tail expression⟩ ⟨tail expression⟩)
  (`if` ⟨expression⟩ ⟨tail expression⟩) （else 節省略の時）

例えば，

```
1: (define f (lambda (x)
2:             (if (list? x) (f (cdr x))
3:                           x))
```

といったプログラムでは，2 行目の再帰的な関数 `f` の呼出しと，3 行目の `x` の評価は末尾位置にある式である．末尾位置には，末尾位置の式の継続と，その関数の呼出し式の継続が一致するという特徴がある．そのため，末尾位置での再帰呼出しは，継続が 2.1 節で定義した順序で真に大きくならない実装が可能で，無限に繰り返すことを可能にできる．さらに，一部の組み込み関数は，別の関数を末尾呼出しする．後述の `call/cc` は，関数呼出し式の継続をキャプチャする．`call/cc` は引数に与えられた関数を末尾呼出しする．なお，Scheme 言語では，上記のプログラムのような関数定義を，

```
(define (f x) ... )
```

と略記できる．

Scheme 言語では，`call/cc` 関数によってキャプチャする．

(call/cc *proc*)

ここで，*proc*は継続を引数に受け取る1引数の関数である．call/cc関数は，引数に与えられた*proc*を末尾呼出しする．これと同時にその関数呼出しの結果を待つ継続をキャプチャし，生成した継続オブジェクトを*proc*の呼出しの引数として渡す．Scheme言語では，継続オブジェクトは1引数関数と同じく次の構文で呼び出すことができる．

(*continuation value*)

*continuation*の評価結果である継続が呼び出されると，その継続を生成したcall/ccの続きに制御が移る．この時，call/cc式の評価結果は*value*の評価結果となる．つまり，*value*を評価した結果を返り値として，call/cc式からリターンすることになる．

次に，Scheme言語で一級継続を使う例を示す．

```
> (define CONT #f)
> (define (f k) (set! CONT k) (k 1) 2)
> (+ (call/cc f) 5)
=> 6
> (CONT 3)
=> 8
```

この例では，関数fの呼出し式の継続をキャプチャしている．生成された継続オブジェクトは関数fに渡され，引数kにバインドされる．関数fではその継続オブジェクトを大域変数CONTに代入し，続いてその継続オブジェクトを1，を引数に呼び出している．その結果，直ちに関数fの呼出し式の継続に1が渡され，5を足した値の6が全体の計算結果となる．もし，継続オブジェクトの呼出しをしていなければ，関数fは最後まで実行され，最後に評価した式の値である2が関数fの呼出し式の継続に渡される．この例では，さらに関数fからリターンした後に，再度，大域変数CONTに保存しておいた継続オブジェクトを呼び出している．これにより，関数fの呼出し式の継続が再度実行され，今度は，継続に渡された引数3に5が足されて8が計算結果となる．

ところで，継続の呼出しにより移動するのは制御だけで，変数の値は継続の呼出しにより変化しない．次に，Scheme言語で代入を行うset!式を使った例を示す．set!式は，

(set! *var expression*)

の構文により，変数*var*に*expression*を評価した結果を代入する．

```
1: (define (id k) k)
2: (define (f x)
3:   (let ((loop (call/cc id)))
4:     (display x)
5:     (set! x (+ x 1))
6:     (loop loop)))
```

ここで，let式は局所変数を導入するScheme言語の構文である．

(let ((*variable init-expression*) ...)
  expression ...)

変数 *variable* を *init-expression* の評価結果を初期値として導入し，本体である *expression* の列を逐次的に評価する．*variable* のスコープは *expression* の列の中だけである．なお，let式では同時に複数の局所変数を導入できる．また，displayは引数を画面に印字する関数とする．上記の例は，整数を受け取り，その値から始まる，1ずつ増える数列を印字する関数fの定義である．3行目で，恒等関数idの呼出しの継続をキャプチャして，継続オブジェクトを局所変数loopにバインドしている．次に，4行目で引数がバインドされている変数xの値を印字し，5行目でxに次の整数を代入している．さらに，6行目でloopにバインドされた継続オブジェクトを呼び出している．loopに格納された継続オブジェクトは，loopに値をバインドした後，4行目を実行するという計算を持っている．したがって，loopにバインドされた継続オブジェクトに同じ継続オブジェクトを渡して呼び出すことで，ループ構造が形成される．さらに，継続呼出しによって変数の値が変化しないため，継続呼出し後も変数xの値は次に印字すべき値を保っている．これにより，関数fは受け取った整数から始まる1ずつ増える数列を無限に印字し続ける関数となる．

2.3 一級継続を使ったプログラミング

一級継続を使えば，あらゆる制御がユーザレベルで実現できる．ここでは，非局所脱出の例である例外処理機構と，マルチタスクの例であるコルーチンを一級継続を使って実現する例を示し，一級継続を持つプログラミング言語の記述性の高さを示す．

2.3.1 例外処理機構

例外処理機構とは，計算の実行中にエラーなどの例外的な事象が発生した場合，実行を中断して，エラーから回復するためのハンドラに制御を移す機能である．関

```
struct list {
  int value;
  list* next;
};
int sumlist_aux(list* p, int acc) {
  if (p == NULL)
    return acc;
  return sumlist_aux(p->next, acc + p->value);
}
void print_sumlist(list* p) {
  int sum;
  sum = sumlist_aux(p, 0);
  printf("%d\n", sum);
}
```

図 2.1: リスト要素の和を求めるプログラム

try 文:

```
 try {
   body
 } catch (Class var) {
   handler-body
 }
```

throw 文:

```
 thorw exception;
```

図 2.2: try 構文と throw 構文

数呼出しの深い位置でエラーが発生した時，計算を中断するためには，非局所的な制御の移動が必要となる．

例えば図2.1のC言語のプログラムを考える．print_sumlistは，整数のリストが与えられ，その要素を足し合わせた値を表示する関数である．リストを再帰的にたどりながら計算を進めるために，sumlist_aux関数を再帰的に呼び出している．このプログラムでは，sumlist_aux関数で足し算をする際に，オーバフローする可能性がある．もしオーバフローすると，それ以降の計算をしても正しい答えは得られないため，直ちに計算を中断し，エラーに対処することが望ましい．

Java言語では，例外処理機構を使ってエラー発生時に処理を中断し，エラーに対処することができる．Java言語の例外処理機構では，try文を使って例外的な事象に対応するルーチンである例外ハンドラを設置し，throw文により例外を発生させて例外ハンドラへ制御を移す．図2.2にtry文とthrow文の構文を示す．try文

```
int sumListAux(List p, int acc) throws OverflowException {
  if (p == null)
    return acc;
  if (checkOverFlow(acc, p.value))
    throw new OverflowException();
  return sumListAux(p.next, acc + p->value);
}
int printSumList(List p) {
  try {
    int sum;
    sum = sumListAux(p, 0);
    System.out.println(sum);
  } catch (OverflowException exception) {
    // Exception handler
    System.out.println("オーバフローしました.");
  }
}
```

図 2.3: 例外処理機構を使ってリスト要素の和を求めるプログラム

では，構文的に囲む式 *body* と，そこから呼び出された関数内の式（これらを try が**動的に囲む**式と呼ぶ）で例外的な事象を検出した時，直ちに例外ハンドラに処理を移すことができる．プログラムは，例外的な事象を検出すると，thorw 文により後述する例外オブジェクトを投げればよい．これにより，*body* 全体の処理は中断され，例外ハンドラに処理を移すことができる．例外ハンドラは，try 文に付随する catch 節で定義する．一つの try 文には複数の catch 節を指定することができる．Java 言語では，java.lang.Throwable クラスかそのサブクラスのインスタンスが例外オブジェクトとなる．それぞれの catch 節は，処理する例外の種類を *Class* で指定する．thorw により例外オブジェクトが投げられた時，その式を動的に囲む最も内側の try 文が，投げられた例外オブジェクトを処理できる catch 節を持っていれば，その catch 節に制御が移る．もし，投げられた例外オブジェクトを処理できる catch 節を持っていなければ，その try 文を動的に囲む try 文の catch 節がのうちから，適切な catch 節が選ばれる．catch 節の実行が終わると，その catch 節を持つ try 文の続きに制御が移る．

これを使うと，print_sumlist は図 2.3 のようなオーバフローに対処したプログラムにできる．ここで，checkOverFlow 関数はオーバフローするかどうかを調べる関数とする．オーバフローすることが分かると，OverflowException クラスの例外を生成して，投げる．これにより制御が printSumList 内の例外ハンドラに移り，エラーメッセージが表示される．

```
(define (sumlist-aux k p acc)
  (if (null? p)
      acc
      (if (check-overflow acc (car p))
          (k 'overflow)
          (sumlist-aux (cdr p) (+ acc (car p))))))
(define (print-sumlist p)
  (let ((exception (call/cc
                    (lambda (k)
                      (let ((sum (sumlist-aux k p acc)))
                        (display sum)
                        'no-exception)))))
    (if (eq? exception 'overflow)
        (display "オーバフローしました."))))
```

図 2.4: 一級継続を使ってリスト要素の和を求めるプログラム

このプログラムを Scheme 言語でファーストクラスの継続を使って記述すると，次のようになる．print-sumlist 関数から sumlist-aux を呼び出す際，その呼出し式の継続をキャプチャして，sumlist-aux に渡している．sumlist-aux は再帰呼出しによりリストをたどりながら計算する．check-overflow 関数でオーバフローを検出すると，print-sumlist 関数から sumlist-aux 関数が呼び出された時の継続を呼び出す．これにより，制御が print-sumlist 関数に戻る．この時継続に'overflow を渡しており，print-sumlist ではこれを調べてエラーメッセージを表示する．このようにして，一級継続で例外処理機構を実現することができる．

2.3.2 コルーチン

コルーチンとは，二つのルーチンが協調して計算を進める計算のパターンである．二つのタスクがあり，片方がデータを生成し，他方が生成されたデータを処理する Producer-consumer 型のマルチタスクはこのパターンの例である．Java 言語では，コルーチンはスレッドを使うことで記述できるが，Scheme 言語では，やはり一級継続を使うことで記述できる．

ここでは，葉要素のみが値を持つ木構造のデータ構造から，葉要素の持つ値のリストを作るプログラムを例題とする．葉要素の値は，リスト上では，木の左から右にたどった順に現れるとする．図 2.5 に問題の例を示す．この例題のプログラムを，再帰呼出しにより木構造を探索するプロデューサタスクと，リストを作るコンシューマタスクが協調して計算を進めるコルーチンのプログラムとして作る．

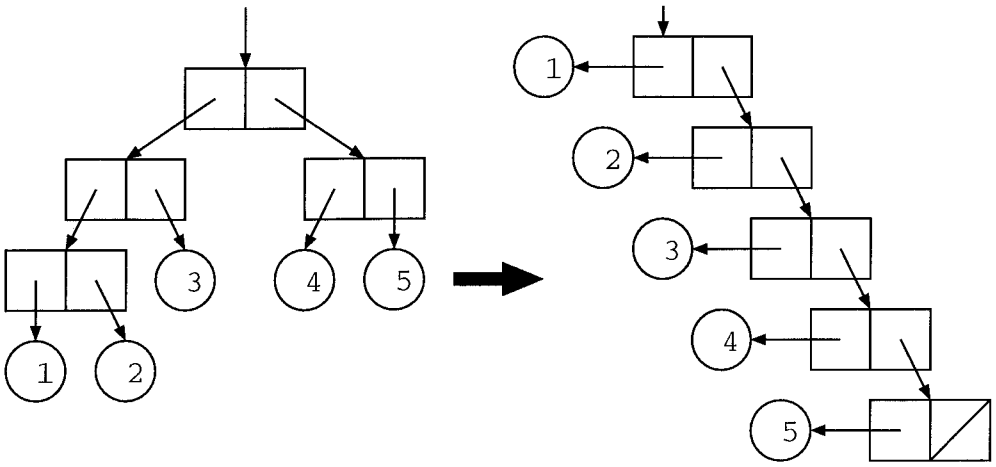

図 2.5: 木構造からリスト構造への変換問題の例

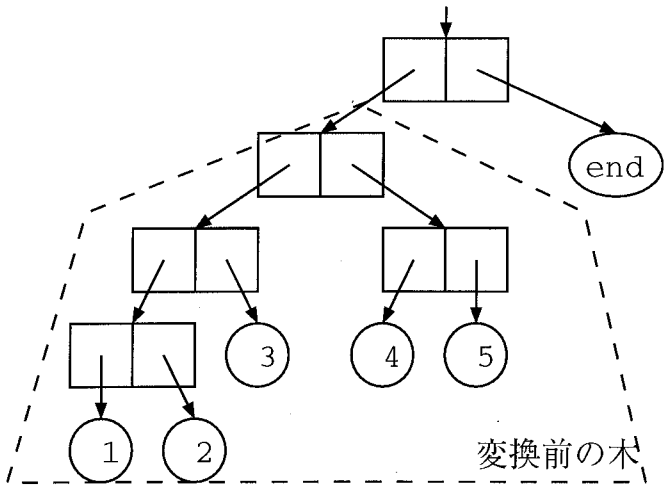

図 2.6: 右端に特殊な値を持つよう変換された木

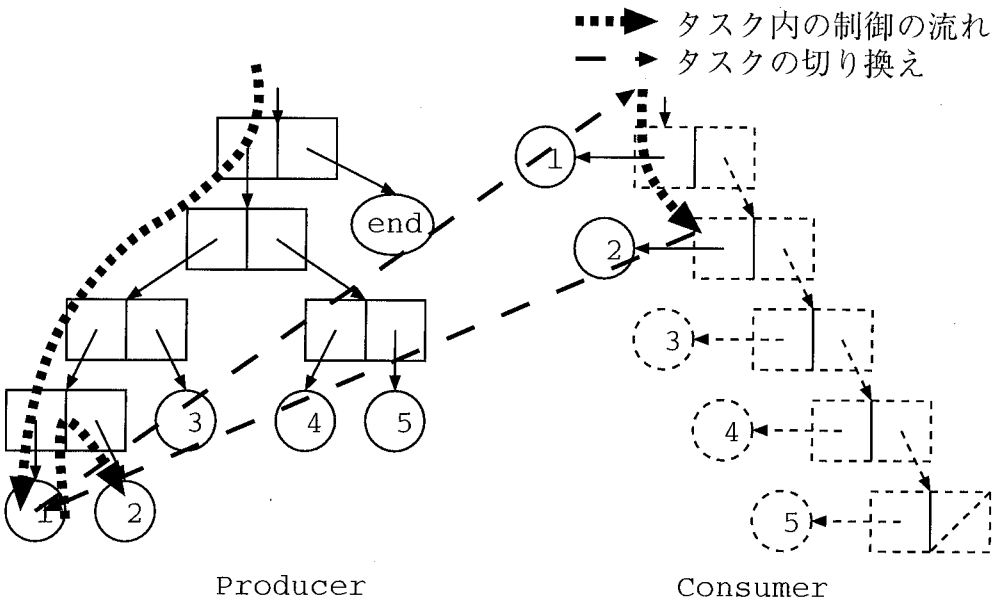

図 2.7: 木構造からリスト構造への変換過程

アルゴリズムを説明する．まず前処理として，図2.6のように木の右端に特殊な値 end を持つ葉を追加する．次に，プロデューサタスクは，左端の葉まで探索を進める．最初の葉が見付かると，それをコンシューマタスクに伝え，制御を渡す．コンシューマタスクはリストの最初の要素を受け取り，次の要素を計算するために，再びプロデューサタスクに制御を返す．プロデューサタスクが end を発見すると，コンシューマタスクにそれを伝えて制御を渡す．コンシューマタスクは，これまでに受け取った値を，逆順にリストに組み上げる．図2.7は，プロデューサタスクが二番目の要素にたどり着くまでの制御の流れを示している．

このアルゴリズムを Java 言語で記述した例を図2.8に示す．execute メソッドにより与えられた木構造をリストに変換するプログラムである．プロデューサタスクは start メソッドにより開始されるスレッドで，run メソッドから実行が始まる．run メソッドは，直ちにプロデューサタスクの本体である producer メソッドを呼び出す．コンシューマタスクは consumer メソッドで，execute メソッドを呼び出したスレッドにより実行される．synchronized 構文により，プロデューサタスクとコンシューマタスクは，同時には片方しか動かない．タスクの切り替えは，待ち状態のスレッドを実行可能状態にする notifyAll メソッドと，自ら待ち状態になる wait メソッドにより実現した．また，スレッド間の通信には，インスタンス変数を使った．このプログラムでは，プロデューサタスクとコンシューマタスクを別のスレッドが実行するため，双方のタスクで関数呼出しを使って探索やリストの生成を簡潔に記述できている．

同様の構造のプログラムは一級継続を使っても記述できる．Scheme 言語で記述

```
class TreeToList extends Thread {
  Tree   tree;
  Object currentValue;
  static Object end = new Object();

  // producer thread
  public void run() {
    try { synchronized (this) {
      producer(new Node(tree, new Leaf(end)));
    } } catch (InterruptedException e) {}
  }
  void producer(Tree tree) {
    if (tree instanceof Node) {
      producer(((Node) tree).left);
      producer(((Node) tree).right);
    } else if (tree instanceof Leaf) {
      currentValue = ((Leaf) tree).value;
      notifyAll();
      if (currentValue == end) return;
      wait(); // switch to the consumer
    }
  }
  List consumer() {
    notifyAll(); wait(); // switch to the producer
    Object value = currentValue;
    if (value == end)
      return null;
    else
      return new List(value, consumer());
  }
  public List execute(Tree tree) {
    try { synchronized(this) {
        this.tree = tree;
        start(); // launch the producer thread
        return consumer();
    } } catch (InterruptedException e) { return null; }
  }
}
```

図 2.8: マルチスレッドによるコルーチンの例

```
(define (producer consumer tree)
  (if (pair? tree)
      (let ((consumer (producer consumer (car tree))))
        (producer consumer (cdr tree)))
      (call/cc
        (lambda (k)
          (consumer (cons k tree))))))  ; switch to the consumer

(define (consumer tree)
  (let loop ((producer&value (call/cc
                               (lambda(k)
                                 (producer k (cons tree 'END))))))
    (let ((producer (car producer&value))
          (value    (cdr producer&value)))
      (if (eq? value 'END)
          '()
          ;; switch to the producer
          (cons value (loop (call/cc producer)))))))

(define (tree->list tree)
  (consumer tree))
```

図 2.9: 一級継続によるコルーチンの例

した，一級継続を使って同様の構造を実現したプログラムを図2.9に示す．このプログラムは一級継続を使って擬似的に二つのスレッドを実現している．プロデューサ，コンシューマとも，他方に制御を渡す時には，まず，自分のタスクの継続をキャプチャし，継続オブジェクトを他方に渡している．プロデューサからコンシューマを呼び出す時は，継続オブジェクト以外に葉要素が持っていた値も渡す必要があるため，二つの値のペアを生成して渡している．一級継続を使えば，マルチスレッド機能を持つプログラミング言語と同等の記述も可能になる．

以上のように，一級継続は様々な制御の流れを表現できる．この点で，一級継続を持つプログラミング言語は，高い記述性を持つ．

2.4 実装戦略

プログラムは関数呼出しを行うことで実行される．関数が呼び出されると，引数や局所変数を格納するための関数フレームが生成される．実行中の関数の関数フレームを**カレントフレーム**（current frame）と呼ぶ．関数の実行中にさらに関数呼出しをすると，新しい関数フレームが生成され，新しい関数フレームがカレントフレームとなる．カレントフレームの内容は常に参照され，また，書き換えられる可能性がある．関数からリターンすると，呼出し元の関数の関数フレームがカレントフレームになる．カレントフレームや，リターンによりまだカレントフレームになる可能性が残っている関数フレームは**アクティブ**であると呼ぶことにする．一級継続を持たないプログラミング言語では，関数からリターンし，関数フレームがアクティブでなくなると，二度と参照されることはなくなる．このような言語の処理系では，関数フレームをスタック上に生成し，管理すると効率がよい．スタックを用いれば，アクティブな関数フレームの領域が常に連続し，アクティブでなくなった関数フレームの領域は直ちに再利用される．

しかし，無限の寿命を持つ一級継続を持つ言語では，一度アクティブでなくなった関数フレームが再びアクティブになることがある．継続の持つ計算は，複数の関数の実行から成っている．これらの関数はまだ実行の途中で，別の関数の呼出しの結果を待っている状態である．継続の実行によりこれらの関数にリターンし，その関数フレームがカレントフレームとなる可能性がある．つまり，継続の実行には，キャプチャした時のアクティブな関数フレーム全体が必要となる．そのため，関数フレームをスタック上に生成する処理系では，関数フレームがアクティブでなくなった時に再利用されてしまい，そのままでは一級継続を利用できない．

そこで，様々な無限の寿命を持つ一級継続の実装戦略が提案されている．ここでは，それらの実装戦略のうち代表的な，GC戦略，スタック戦略，インクリメンタル・スタック/ヒープ戦略を紹介する．

なお，William D. Clingerはプログラムの典型的な継続の利用法は，次のようになものであるとしている[9]．継続の実装を評価する際に，この点についても着目する．

一級継続を使わないシナリオ 多くのプログラムは一級継続を一切使わない．したがって，一級継続を提供する言語であっても，それを使っていないプログラムにはオーバヘッドがないことが望ましい．文献[9]では，一級継続を一切使っていなプログラムで，一級継続のないスタックを使った普通の処理系と比べて，関数の呼出しとリターンに余分なオーバヘッドがない場合，その実装は**ゼロオーバヘッド**（zero overhead）であると呼んでいる．

非局所脱出 Scheme言語で最もよく使われている一級継続の利用方は，例外処理のような非局所脱出である．非局所脱出のための継続は，その継続がアクティブな間に一度だけ呼び出されるか，例外を発生させる可能性のある計算が終わるまで一度も呼び出されないかのどちらかしかない．

頻繁にキャプチャするシナリオ Oliver Danvyによれば，継続のキャプチャは集中して起こる傾向にある[13]．つまり，一度継続がキャプチャされると，その継続を含む継続やその継続に含まれる継続，また，継続の大部分を過去にキャプチャした継続と共通に含む継続が再びキャプチャされる可能性が高い．一級継続を使ってループやコルーチンを実装する場合などがこれに当たる．

2.4.1 GC戦略

GC戦略（garbage collection strategy）[9, 5]は無限の寿命を持つ一級継続の実装戦略として，最も単純な戦略である．GC戦略を実装した処理系には，Standard ML/New Jersey[5]がある．この戦略では，関数フレームをスタックを使って管理せず，ヒープ上に確保する．ヒープ上に確保された関数フレームは，サボテン状に動的リンク（dynamic link）でつながれる．ヒープ上の関数フレームは，通常のオジェクトと同様に，ごみ集めの対象となり，不要になった後に自動的に回収，再利用される．動的リンクは，呼び出された関数の関数フレームから呼び出した関数の関数フレームに向けて参照しているため，通常は，カレントフレームからサボテンの根元までの経路状にあるアクティブな関数フレーム以外はどこからも参照されず，ごみ集めで回収される．しかし，継続をキャプチャすると，継続オブジェクトからカレントフレームへの参照が作られる．図2.10に，GC戦略の処理系の関数フレームと継続オブジェクトの様子を示す．図のように，今後アクティブになる可能性が残っている関数フレームは，継続オブジェクトかカレントフレー

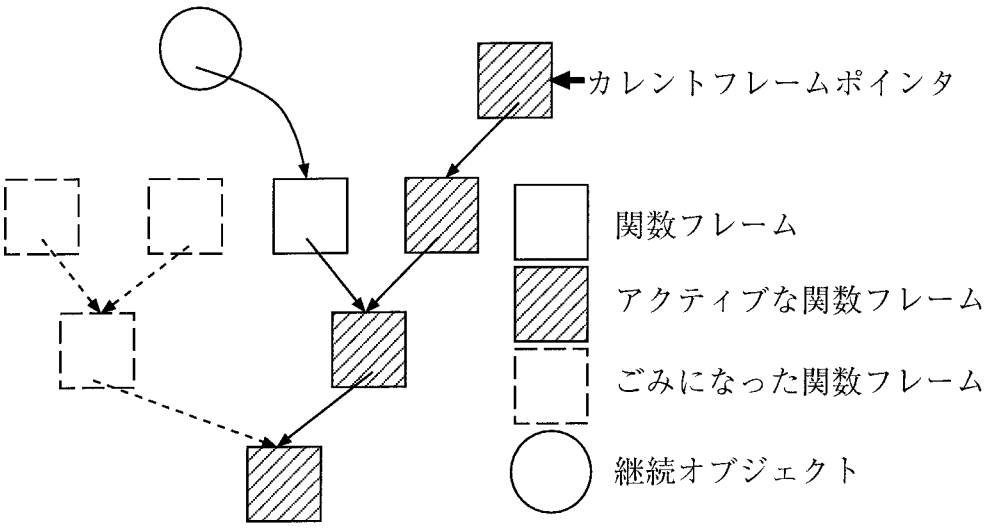

図 2.10: GC 戦略

ムポインタ（current frame pointer）からたどり着くことができるようになっている．それ以外の関数フレームはどこからも参照されず，ごみとして回収される．

GC 戦略は，一級継続を使わないプログラムでも，関数呼出しやリターンに余分なオーバヘッドがあり，ゼロオーバヘッドな戦略ではない．関数呼出しの際，スタックベースの処理系ではスタックポインタを移動させるだけで関数フレームを確保できる．これに対して，GC 戦略では，呼び出される関数の関数フレームをヒープ上に割り当てる．そのため，割り当てに時間がかかる可能性がある．また，関数からリターンすると，スタックベースの処理系ではスタックポインタを移動させ，直ちに関数フレームのメモリ領域を解放するのに対し，GC 戦略では，後の GC に解放をゆだねる．そのため，関数からリターンする度にごみが増え，ごみ集めが頻繁に起こるようになる．割り当てと回収が高速なごみ集めを持っていない処理系では，ごみ集めが頻繁に起こることによる性能低下は著しい．

継続のキャプチャについては，継続オブジェクトから関数フレームへのポインタを作るだけでよいため効率がよい．また，継続の呼出しも，カレントフレームポインタが，継続が持つ関数フレームを指すようにするだけで良いので効率がよい．GC 戦略は，頻繁に継続をキャプチャするシナリオでは効率良く動作する．

2.4.2 スタック戦略

スタック戦略（stack strategy）は，SCM[14] や MzScheme[15] といった Scheme 処理系を始め，様々な処理系で採用されている．スタック戦略の特徴の一つに，スタックベースの処理系に非常に簡単に実装できるという点がある．

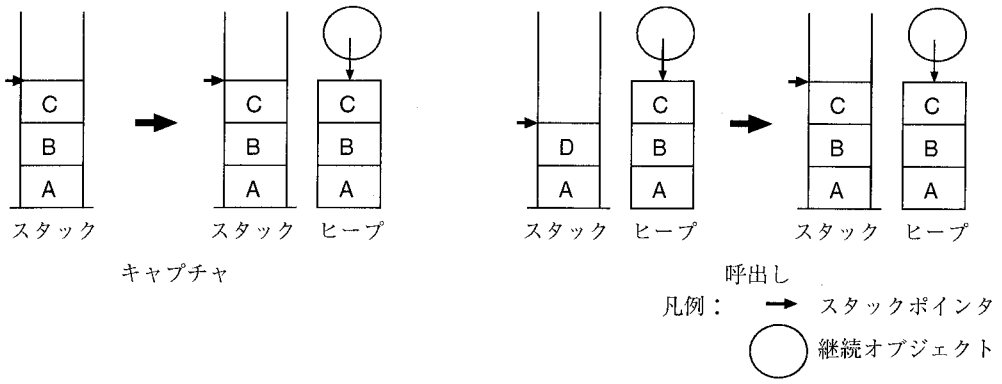

図 2.11: スタック戦略による継続のキャプチャと呼出し

スタック戦略では，一級継続を持っていない処理系と同様に，関数フレームをスタック上に割り当てる．継続をキャプチャすると，スタックの内容をそのままヒープにコピーし，将来の呼出しに備える．アクティブな関数フレームはすべてスタックに積まれているため，スタック全体の内容，つまり，スタックボトムからスタックポインタの指すアドレスまでをコピーすることで，必要な関数フレームは保存される．継続が呼び出されると，ヒープ上にあるスタックのコピーがスタックに書き戻されて，実行が再開する．図2.11にスタック戦略による継続のキャプチャと呼出しの様子を示す．継続のキャプチャでは，スタック全体をヒープにコピーし，継続オブジェクトからの参照を作る．その継続オブジェクトが呼び出されると，ヒープ上のコピーをそのままスタックにコピーしスタックポインタを調整する．

この戦略は，スタックの構成が詳細に分からなくても実装できる．例えば，C言語はスタックの構成を詳細に知ることができない言語である．C言語では，スタックやレジスタを直接参照したり書き換えたりすることはできない．これにより，ソースコードがプロセッサなどのハードウェアに依存しない，高いポータビリティが実現できている．しかし，C言語の関数フレームを高級言語の関数フレームとしても使っている高級言語のインタプリタや，高級言語からC言語に変換するコンパイラでは，C言語のスタックやレジスタが直接参照できないため，GC戦略はもちろんのこと，後述するインクリメンタル・スタック/ヒープ戦略など，スタック戦略以外の実装戦略が使えない．スタック戦略を使えば，次のようにして，C言語で記述されたインタプリタで一級継続を実装できる．

スタックポインタ C言語では，スタックポインタの値を直接取得することはできないが，局所変数のアドレスが得られる．これを使えば，少なくともカレントフレームのどこかを指すアドレスが間接的に得られる．スタック戦略で継続をキャプチャする時は，少なくともスタックボトムとスタックポインタの

指すアドレスの間はヒープにコピーしなければならないため，継続をキャプチャする位置でダミーの関数を呼出し，ダミーの関数内の局所変数のアドレスでスタックポインタの値を代用する．

スタックボトム スタックボトムには，高級言語の処理を始める前のスタックポインタの値を使えばよい．つまり，高級言語の処理を始める前に，局所変数のアドレスを得て，これをスタックボトムとすればよい．その後，高級言語の解釈実行をする関数を呼び出せば，高級言語の解釈実行をしている関数フレームは，最初に得たスタックボトムのアドレスよりスタックの上位方向に作られる．

その他の制御情報 継続のキャプチャでは，関数フレーム以外に，プログラムカウンタやスタックポインタなどの制御情報も保存しなければならない．C言語では，`setjmp`により制御情報を保存し，`longjmp`により保存してあった状態に戻すことができる．

このように，C言語で記述されたインタプリタでは，スタックのコピーと`setjmp`，`longjmp`を組み合わせることで一級継続を実現できる．高級言語からC言語に変換するコンパイラでも同様の手法で一級継続を実現できる．

次に，スタック戦略の特徴を議論する．スタック戦略はゼロオーバヘッドな実装戦略である．継続を一切キャプチャしないプログラムでは，一級継続を持たないスタックベースの処理系と同じ方法で関数呼出しとリターンをする．また，前述のように，関数フレームの構造が正確に分からない場合でも利用でるという利点がある．さらに，関数フレームがリロケータブルである必要がないという利点もある．この点についてもう少し詳しく説明する．C言語などでは，関数フレームのダイナミックリンクなどに関数フレームの絶対アドレスを使っている．このような場合，もし継続の呼出し後，関数フレームが最初に生成された場所とは別の場所に復元されると，正しくリターンすることができない．また，関数フレーム内の局所変数のアドレスがどこかに格納され，継続の呼出し後に，局所変数が間接参照される場合も同様の問題がある．スタック戦略では，関数フレームは最初に生成したアドレスに復元されるため，このような問題は起こらない．

一方，継続のキャプチャや呼出しではスタック全体のコピーを行うので，非常に時間がかかる．さらに，スタックは一般のオブジェクトに比べて非常に大きく，関数の呼出しの深さに比例していくらでも大きくなるため，スタックのコピーがヒープの大きな領域を占有する．その結果，一般のオブジェクトに利用可能なヒープが狭くなり，ごみ集めが頻繁に起こるようになる．この状況は，継続が頻繁にキャプチャされるシナリオではさらに顕著になる．

SOAR[16] は，専用のハードウェアを使うことで，非局所脱出のシナリオでスタックのコピーなしに継続のキャプチャと呼出しができるようにしたSmalltalk-80言語の処理系である．SOREについては，第4章の関連研究でとりあげる．

2.4.3 インクリメンタル・スタック/ヒープ戦略

Larceny[10] やScheme48[11] で使われている**インクリメンタル・スタック/ヒープ戦略**（incremental stack/heap strategy）[9] は，一級継続を使わないプログラムでは一級継続を持たないスタックベースの処理系と同様に動作する．しかし，いったん継続がキャプチャされると，アクティブな関数フレームをヒープにコピーし，スタックを空にする．ヒープにコピーした関数フレームは，GC戦略と同様に，動的リンクでつながったリストになり，継続オブジェクトがリストの先頭への参照を持つ．また，ヒープにコピーしたアクティブな関数フレームの先頭は，処理系の持つレジスタで保持しておく．このレジスタを，ここではmoreレジスタと呼ぶことにする．Scheme言語では，`call/cc`により，関数呼出し式の継続をキャプチャする．したがって，呼び出される関数は，空になったスタックの底に関数フレームを生成し動作することになる．`call/cc`により呼び出された関数からリターンすると，スタックアンダーフローが起こる．これを検出して，moreレジスタの指す本来のリターン先の関数の関数フレームをスタックの底に書き戻し，リターン先の関数を実行する．この時同時に，moreレジスタが，スタックの底の関数フレームのリターン先となる関数フレームを指すように更新しておく．これは，moreレジスタが指していた関数フレームのリストの二番目の要素である．継続オブジェクトの呼出しも，スタックアンダーフローと共通の動作になる．継続の呼出しでは，呼び出された継続オブジェクトが持つ関数フレームのリストをmoreレジスタに設定する．これにより，継続オブジェクトの持つ関数フレームのリストがアクティブになる．さらに，スタックを空にして，スタックアンダーフローを発生させると，継続の実行が始まる．

インクリメンタル・スタック/ヒープ戦略では，一度継続をキャプチャすると，アクティブな関数フレームはヒープに移動される．そのため，二回以降の継続のキャプチャでは，アクティブな関数フレームのうち，前回のキャプチャ以降に生成されて，まだスタックに残っている関数フレームしかコピーされない．一度コピーされた関数フレームは，ヒープ状のリストが共有され，再びコピーされることはない．このように，インクリメンタル・スタック/ ヒープ戦略では，継続のキャプチャによって寿命の延びた関数フレームだけでヒープ上に関数フレームのサボテン構造が作られる．

図2.12はインクリメンタル・スタック/ ヒープ戦略を実装した処理系のスタッ

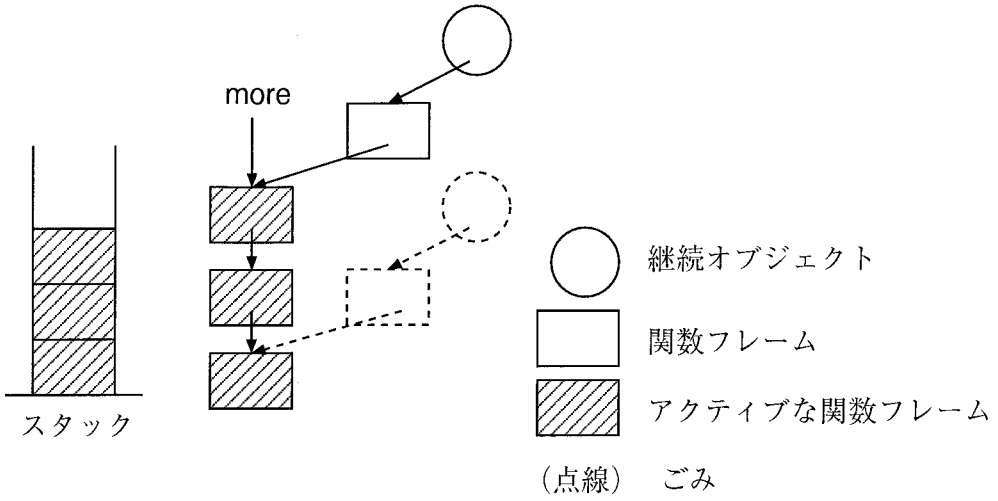

図 2.12: インクリメンタル・スタック/ヒープ戦略

クと関数フレームを模式的に表した図である．図のように，アクティブな関数フレームのうち，最後に継続がキャプチャされ，あるいは呼び出されて以降に生成された関数フレームはスタック上に残っているが，残りのアクティブな関数フレームはヒープにあり，moreレジスタから指されるリストになっている．また，継続オブジェクトも関数フレームのリストを持っており，リストの末尾部分はmoreレジスタが指すリストや継続オブジェクトの持つリストで共有されている．継続がごみとなると，その継続オブジェクトからしか参照されていなかった関数フレームも同時にごみになり，ごみ集めにより回収される．

インクリメンタル・スタック/ヒープ戦略は，スタックアンダーフローを検出して，リターン先の関数フレームをヒープからスタックにコピーする．そのため，スタックアンダーフローを効率良く検出する仕組が必要になる．これは，スタックボトムの関数のリターンアドレスを,「リターン先の関数フレームをヒープからスタックにコピーし，その関数に制御を移す」という処理のコードへのアドレスにしておけばよい．

次に，インクリメンタル・スタック/ヒープ戦略の特徴を議論する．インクリメンタル・スタック/ヒープ戦略も，ゼロオーバヘッドな実装戦略である．さらに，継続が頻繁にキャプチャされるシナリオでも効率良く動作する．これは，二回目以降の継続のキャプチャで，以前にヒープに移動された関数フレームを新しく生成する継続オブジェクトでも共有して利用するためである．頻繁に継続がキャプチャされる場合，連続する継続のキャプチャの間で新しく生成される関数フレームはそれほど多くない．そのため，毎回すべての関数フレームをコピーするスタック戦略に比べて，格段に少ない数の関数フレームのコピーで継続のキャプチャが実現できる．

しかし，一度ヒープに移動された関数フレームにリターンする際には，明示的な継続の呼出しによる制御の移動ではない通常のリターンであっても，ヒープからスタックに関数フレームを書き戻さなければならない．継続のキャプチャ後の実行効率はやや悪くなる．

最後に，インクリメンタル・スタック/ヒープ戦略の実装上の問題について触れる．この戦略の実装は関数フレームの構造に依存する．関数フレームをスタックからヒープにコピーした後は，関数フレームのリストとして保持する．これには，スタック上での関数フレームの境界が正確に分かっていなければならない．また，スタックアンダーフローの際に関数フレームをスタックボトムに書き戻すが，その関数フレームのリターンアドレスを，次のスタックアンダーフローに備えて特殊なアドレスに書き換える．これも，関数フレーム中のリターンアドレスの位置が分からなければ実現できない．また，ヒープにコピーされた関数フレームは実行される時にスタックボトムに書き戻されるが，これは一般に，最初にその関数フレームが生成されたアドレスとは異なる．そのため，関数フレームや関数フレーム内に割り当てられた変数やオブジェクトの絶対アドレスを使う処理系では実装できない．このように，インクリメンタル・スタック/ヒープ戦略は，どのような処理系にでも実装できるわけではない．特にC言語の制御スタックを高級言語の制御スタックとしても利用するインタプリタや，高級言語からC言語に変換するコンパイラで利用できない点は応用の範囲を狭めている．

第3章　他言語との相互呼出し可能な処理系のための実装

「ぶぶ」[17] は Java 言語で記述した Scheme 処理系である．ぶぶは，ぶぶオブジェクトシステム [18] という機構を持ち，Java 言語との言語透過性を実現している．ぶぶオブジェクトシステムを用いることで，Scheme 言語で記述された関数と Java 言語で記述されたメソッドが相互に呼び出し合うプログラムを記述することができる．このようなプログラムの Java 言語で記述されたメソッドは，Java バーチャルマシンで直接実行される．Scheme 言語は継続を無限の寿命を持つ一級オブジェクトとして扱う機能を備えており，制御スタックをぶぶの処理系から自由に操作する必要がある．しかし，Java 言語で記述されているぶぶの処理系は，Java バーチャルマシンで直接実行されている Java 言語で記述されたメソッドのスタックを操作することはできない．

この章ではこのような処理系に，完全ではないが多くの場面で有効に利用できる一級継続を実装する手法を提案する．合わせて，Java の例外処理機構を Scheme 言語からシームレスに扱う機能や，継続の呼出しにより起る大域的な制御の移動の際に実行する手続きを登録する `with-handler` 関数の実装も行い，一級継続と組み合わせることにより有効に利用できることを示す．まず，3.1 節では，本研究のターゲットとなるぶぶについて詳しく説明する．また，ぶぶ以外の Java 言語で記述された Scheme 処理系も紹介する．3.2 節では，ぶぶに実装する制御機能の設計を行い，3.3 節でその実装方法を示す．その上で，実装した制御機能について考察を 3.4 節で行う．

なお，この章の内容は [19] において発表した内容を含んでいる．

3.1　Java 上の Scheme 処理系

3.1.1　Scheme 処理系ぶぶ

本論文の対象としているぶぶについて説明する．ぶぶは Java 言語で記述された Scheme 言語の処理系である．Scheme 言語の標準仕様 [2, 3] に準拠しており，さらに，Java 言語の機能を利用して次のような拡張を行っている．

- Java 言語で記述されたモジュールを利用する機能.
- Java 言語レベルのスレッドを利用するインタフェース.

ぶぶは完全に Java 言語で記述されているため，Java アプレットに組み込んで，多くの WWW ブラウザ上で実行することもできる.

Java 言語で記述されたモジュールを利用する機能については 3.1.2 節で，Java 言語レベルのスレッドを利用するインタフェースは 3.1.3 節で詳しく説明する.

ぶぶは，基本的にはインタプリタとして動作する．対話的に式を入力されると，式を即座にぶぶバイトコードと呼ぶ形式にコンパイルし，それを解釈実行する．また，ファイルから Scheme 言語で記述されたプログラムを読み込むこともできる．さらに，ぶぶバイトコードの状態でファイルに保存しておくこともできる．これを読み込むことで，毎回コンパイルすることなしに実行することもできる.

3.1.2 オブジェクトシステム

ぶぶには，Scheme 言語に対する拡張機能として，Java 言語で記述されたモジュールを利用する機能が備わっている．この機能を**ぶぶオブジェクトシステム** [18] と呼ぶ．ぶぶオブジェクトシステムは次のような機能からなる.

- Java 言語のオブジェクト（Java オブジェクトと呼ぶことにする）を Scheme 言語の一級オブジェクト（Scheme オブジェクトと呼ぶことにする）として扱う機能.
- Java オブジェクトのインスタンス変数や，Java 言語で定義されたクラス（Java クラスと呼ぶことにする）のクラス変数を読み書きする機能.
- Java オブジェクトのインスタンスメソッドや，Java クラスのクラスメソッドを呼び出す機能.
- Java クラスや Java 言語レベルのインタフェース（クラスとの違いは本論文では重要ではないので，説明のためにクラスとして扱う）を拡張して，Scheme 言語レベルのクラス（Scheme クラスと呼ぶことにする）を定義する機能.
- Java クラスや Scheme クラスからインスタンスを作る機能.

これらの中でも，Java クラスを拡張して Scheme クラスを定義する機能は特に強力である．Java 言語のクラスライブラリの中には，ライブラリが提供するクラスをプログラマが拡張して，いくつかのメソッドをオーバライドするような利用を前提としているものが少なくない．このようなライブラリを利用するためには，

C1.java:

```
public class C1 {
  public Object m1() { return m2(); }
  public Object m2() { return null; }
}
```

Scheme:

```
(defclass C2 (C1))                              ; (1)
(defmethod  C2 m2 () (f2))                      ; (2)
(define (f1) (send (new C2) m1))                ; (3)
(define (f2) (format #t "f2 is called.~%"))
```

図 3.1: Java クラスを拡張して Scheme クラスを定義する例

Java クラスを拡張する機能が必須である．拡張を前提としたモジュールの例として，Java 言語の標準パッケージの java.awt パッケージが挙げられる．java.awt パッケージはウィンドウやボタン，テキスト入力フィールドなどを使ってグラフィカルな入出力を行うためのパッケージである．このパッケージでは，キー入力やマウスクリックなどをイベントとして扱う．イベントが発生すると，あらかじめ登録してあるオブジェクトの，イベントに対応したメソッドが呼び出されるようになっている．java.awt パッケージを使うプログラマは，イベントを扱うための基本クラスを拡張して，処理したいイベントに対応するメソッドをオーバライドする．

ぶぶを使って，Java クラスを拡張して Scheme クラスを定義する例を図 3.1 に示す．

この例では，(1) Java クラス C1 を拡張した Scheme クラス C2 の定義，(2) クラス C2 のメソッド m2 を定義してクラス C1 のメソッドをオーバライド，(3) クラス C2 のインスタンスの生成と生成されたクラス C2 のインスタンスのメソッド m1 の呼出しを行っている．この例の関数 f1 を呼び出すと，クラス C2 のインスタンスが生成され，そのメソッド m1 が無引数で呼び出される．メソッド m1 はクラス C2 ではオーバライドされていないので，クラス C1 のメソッド m1 が呼び出されることになる．メソッド m1 では，メソッド m2 を呼び出すが，これはクラス C2 でオーバライドされているため，クラス C2 のメソッド m2 が呼び出される．メソッド m2 の本体では関数 f2 を呼び出しているため，関数 f2 が実行され，

```
f2 is called.
```

と画面に表示される．

3.1.3 マルチスレッド

ぶぶオブジェクトシステムを用いると，次のような方法を使ってプログラマはJava言語レベルのスレッドを自由に利用することができる．

- Java言語の標準パッケージにあるThreadクラスを拡張してSchemeクラスを定義する．
- Java言語で記述されたスレッドを使うモジュールを，Scheme言語から呼び出す．

このようにして作られたスレッドは，Scheme言語で記述されたメソッドを呼び出すことができるようになっている．

また，より簡単にスレッドを利用することができるようにするために，ぶぶはJava言語レベルのスレッドを扱うインタフェースを持っている．このインタフェースによって作られるスレッドは，Threadクラスを拡張したスレッドのインスタンスであり，ぶぶスレッドと呼ぶ．

3.1.4 処理系の内部構造

ぶぶは，バイトコードインタプリタと呼ばれるぶぶバイトコードを解釈実行する部分と，システムライブラリとで成っている．Scheme言語からぶぶバイトコードに変換するコンパイラや，トップレベルのread-evel-printループなどはライブラリの一部である．入力された式は，対話的に入力された式も含めて，ライブラリによってコンパイルされた後に，バイトコードインタプリタによって解釈実行される．本論文では，ぶぶバイトコードとScheme言語のプログラムを区別しない．

バイトコードインタプリタはスレッドの実体であり，次のインスタンス変数を持つJava言語のオブジェクトである．

スタック 実行のためのスタックである．制御スタックと値を積むスタックを兼ねている．Java言語の配列で実現されている．

スタックポインタ スタックトップのインデックスを保持するレジスタである．

ベースレジスタ 現在実行中の関数の関数フレームの，スタック上での位置を保持するレジスタである．

アクティブレジスタ 現在実行中のアクティブフレームの，スタック上での位置を保持するレジスタである．アクティブフレームとは，トップレベルで定義された関数のフレームと，そこから呼び出されたローカル関数のフレームを合

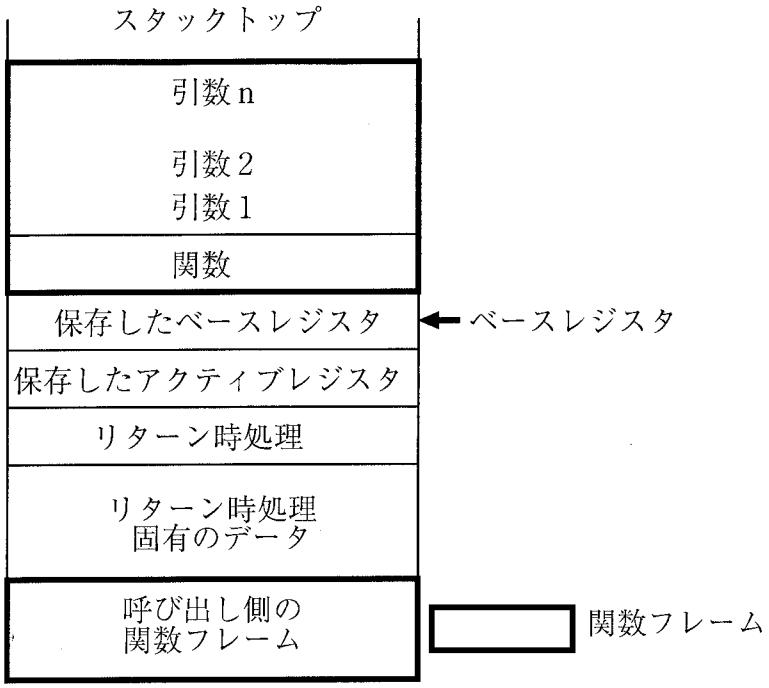

図 3.2: ぶぶの関数フレーム

わせたものである．ローカル関数はアクティブフレーム内の変数を参照する可能性があるため，ぶぶでは関数フレームをアクティブフレーム単位で扱っている．以降アクティブフレームを指して関数フレームと呼ぶ．

アキュムレータ 関数の返り値の受け渡しなどに使われるレジスタである．

プログラムレジスタ 現在実行中のプログラム本体であるぶぶバイトコードの配列を指すレジスタである．

プログラムカウンタ プログラムレジスタが指す配列上の，現在実行中のバイト命令のインデックスを保持するレジスタである．

スタック上の関数フレームは，呼び出された直後は図 3.2 に示す形をしている．関数は必要があれば関数フレームを書き換えながら実行を行う．関数からリターンする際は，ベースレジスタ，アクティブレジスタをスタックから取り出した後に,「**リターン時処理**」を取り出して，その処理を行う．通常の関数呼出しでは，リターン時処理固有のデータにはリターン先のプログラムカウンタが入っており,「リターン時処理」の実行でプログラムカウンタを書き換える．リターン時に特別な処理が必要な関数では，特別な「リターン時処理」を積むことにより，リターン時に様々な処理を行うことができる．

バイトコードインタプリタは，Schemeクラスのメソッドを実行するJava言語レベルのスレッドに1つずつ作られる．Schemeクラスのメソッドが呼び出される時は，そのスレッドに関連付けられたバイトコードインタプリタを検索し，なければ新たに生成するが，既にあればそれを使う．

3.1.5 オブジェクトシステムの実装

ぶぶオブジェクトシステムは，Javaクラスを拡張した動的基本クラス（dynamic base class）と呼ぶJavaクラスを用いることで，Schemeクラスの定義を可能にしている．Schemeクラスは，実装上はクラス定義を保持するJavaオブジェクトである．あるJavaクラスXを拡張したSchemeクラスYを定義するには，Xを拡張した動的基本クラスX′を用いる．動的基本クラスX′はスーパークラスのprivate宣言もfinal宣言もされていないすべてのメソッドをオーバライドしており，意味上のクラスを表すSchemeクラスYへの参照を持っている．SchemeクラスYのインスタンスは実装上は動的基本クラスX′のインスタンスとなっており，メソッドの呼出しは次のように行われる．

- スーパークラスであるJavaクラスXで定義されたメソッドをScheme言語から呼び出す時は，動的基本クラスX′のメソッドを呼び出す．動的基本クラスX′のメソッドでは，そのメソッドがSchemeクラスYでオーバライドされているかどうかを調べる．オーバライドされている場合は対応するScheme言語の関数をバイトコードインタプリタで実行する．そうでなければ，スーパークラスの対応するメソッドを呼び出す．

- スーパークラスであるJavaクラスXで定義されたメソッドをJava言語から呼び出す時も，動的基本クラスのメソッドを呼び出すことになる．これはSchemeクラスYのインスタンスの実体が動的基本クラスX′のインスタンスだからである．動的基本クラスのメソッドはScheme言語から呼び出されたのと同じ動作をする．

- SchemeクラスYで新たに定義したメソッドをScheme言語から呼び出す時は，処理系がSchemeクラスYの定義を調べて，対応するScheme言語の関数をバイトコードインタプリタで実行する．

- Java言語ではコンパイル時に呼び出すメソッドが定義されているかチェックされる．この時Schemeクラスまだ定義されていないので，Schemeクラスで新たに定義したメソッドをScheme言語から呼び出すことはない．

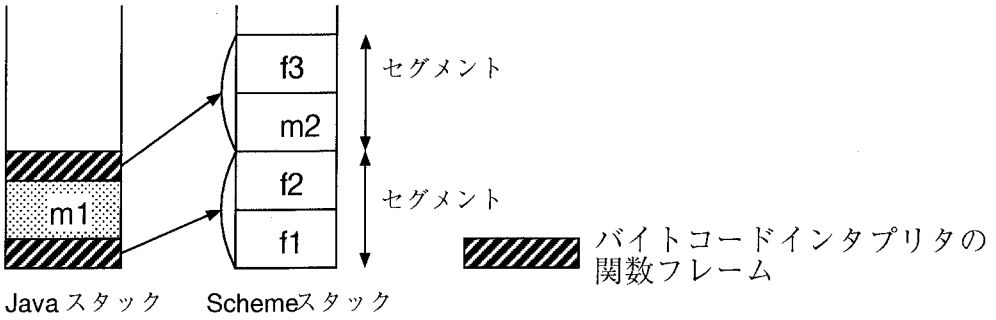

図 3.3: Java と Scheme の相互運用時のスタック

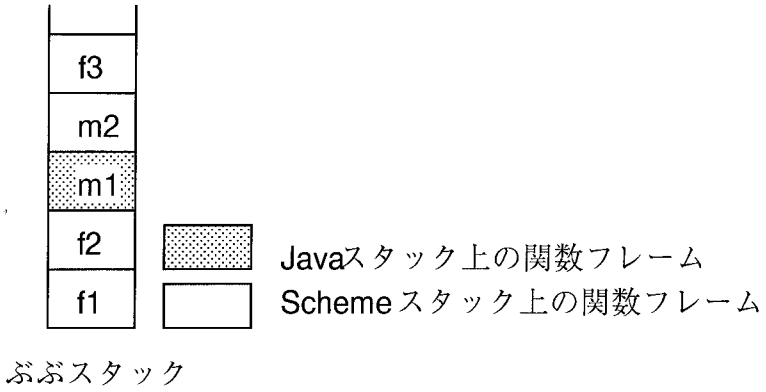

図 3.4: ぶぶスタック

Scheme 言語で記述された関数はバイトコードインタプリタで実行するが，Java 言語で記述されたメソッドは Java バーチャルマシンが直接実行する．また，バイトコードインタプリタ自身も Java バーチャルマシン上で実行される．したがって，一般的な Java バーチャルマシンでは，図 3.5 のプログラムを実行中，処理系の内部では図 3.3 のようになっている．ここで Java スタックとは Java バーチャルマシンが実行に用いるスタックを指し，Scheme スタックとはバイトコードインタプリタが持つスタックを指す．

図 3.3 は，図 3.5 のプログラムの関数 `f1` をトップレベルで呼び出した後，`f3` まで実行が進んだ時の処理系の状態を表している．このように，意味的には図 3.4 のように一本であるスタックを，二本のスタックで実現している．したがって，ぶぶのプログラムの制御は，Scheme スタックと Java スタックの間を行き来することになる．

図 3.3 において，Scheme スタックをメソッド呼出しで区切ったそれぞれを**セグメント**と呼ぶことにする．セグメントは Scheme 言語の機能だけで動作している範囲を指す．セグメントの生成およびセグメントへの出入りは，次のタイミングで行われる．

C1.java:

```
public class C1 {
  public Object m1() { return m2(); }
  public Object m2() { return null; }
}
```

Scheme:

```
(defclass C2 (C1) (:method m2() (f3) ...))
(define (f1) (f2) ...)
(define (f2) (send (new C2) m1) ...)
(define (f3) '())
```

図 3.5: Java と Scheme の相互運用をするプログラム例

- Java 言語レベルのスレッドの実行が開始された時点では，制御はどのセグメントにも入っていない．
- それぞれの Java 言語レベルのスレッドで最初に Scheme 言語で記述された関数が呼び出された時と，Scheme クラスのメソッドまたはコンストラクタが呼び出された時に，セグメントが新しく作られ，そのセグメントに入る．これをセグメントの呼出しという．
- 呼出しによってセグメントが作られた関数の呼出しから制御が戻るとそのセグメントから出る．これをセグメントからリターンするという．
- ぶぶオブジェクトシステムにより，Java クラスまたは Scheme クラスのメソッドを呼び出す時は，その呼出しから制御が戻るまでの間，呼び出した側のセグメントから出る．

3.1.6 他の Scheme 処理系

ぶぶ以外の Java 上の Scheme 処理系としては，Kawa[20] や Silk[21] がよく知られている．Kawa は Scheme 言語で記述されたプログラムを Java バイトコードにコンパイルして Java バーチャルマシンで直接実行する方式をとっている．Silk は Scheme 言語の関数呼出しを Java 言語のメソッド呼出しで実現しているため，ぶぶの Scheme スタックに相当するものは持っていない．Kawa や Silk でも Java 言語で記述されたメソッドを呼び出すことができる．しかし，これらの処理系では，ぶぶのような Scheme クラスを定義することはできず，Java 言語で記述されたメ

ソッドを一方的に呼び出すだけの利用となる．また，一級継続については，Java言語の例外を用いて実現しており，継続オブジェクトの寿命に制限がある．

3.2 設計

本研究では，ぶぶに次の制御機能を実装する．

- 一級継続
- 言語レベルの `dynamic-wind` 関数
- Java言語の例外処理機構をScheme言語からシームレスに扱う機能

これらのうち，一級継続と `dynamic-wind` はScheme言語の仕様で動作が規定されている．しかし，Java言語との相互呼出しが可能なぶぶで，完全な一級継続を，ポータブルな方法で実現するのは不可能である．そこで，実用的で，かつ，論理的に一貫した仕様を検討する．さらに，継続が完全なものでないため，`dynamic-wind` 関数をユーザレベルで一級継続により実現することができなくなる．そのため，`dynamic-wind` 関数を言語レベルで提供する．例外処理機構は，下向きの継続呼出しにより，一級継続で実現できるが，Java言語のメソッドと協調動作するScheme言語の関数を記述するには，Java言語の例外処理機構を扱う機能も必要である．この節では，これらの機能の設計を行う．

3.2.1 例外処理機能の設計

ここでは，Java言語の例外処理機構をScheme言語からシームレスに扱う機能を設計する．つまり，例外ハンドラの設置と例外を投げる式の構文と意味を定める．

例外ハンドラの設置

Java言語では，予期しない状態になったことは例外を用いて呼び出した側に伝えられる．ぶぶオブジェクトシステムを用いてScheme言語からJava言語で記述されたメソッドを呼び出すと，Java言語で記述されたメソッドの実行中に例外が発生する可能性がある．発生した例外はJava言語の仕様では，発生した式を動的に囲む `catch` 節のうち最も内側のものから順に例外ハンドラが検索される．そこで，Scheme言語にも `try` 文に相当する構文を追加し，Scheme言語で記述された関数とJava言語で記述されたメソッドが入り混じったぶぶのプログラムとして見た時に，例外が発生した式を動的に囲む `catch` 節のうち最も内側のものから順に

検索することにする．ただし，Scheme 言語は Java 言語と違い動的なプログラミング言語なので，catch 節はあらかじめ設置しておくのではなく，処理系が随時設置と除去を行うことにする．

Java 言語の try 文に相当する構文として，with-handler 関数を Scheme 言語に追加する．

```
(with-handler ((class1 handler1)
               (class2 handler2)
               ...  )
  body)
```

with-handler は二つの引数をとる．第一引数は catch 節に相当する，例外のクラスのクラスオブジェクト *class* と例外ハンドラ *handler* の組のリストである．*class* は java.lang.Throwable クラスかそのサブクラス（Java クラスでも Scheme クラスでもよい）のクラスオブジェクトでなければならず，*handler* は例外オブジェクトを受け取る 1 引数の関数でなければならない．第二引数の *body* は実行する本体となる関数である．with-handler は catch 節を設置して，*body* を無引数で呼び出す．*body* から普通にリターンすると，catch 節を除去して，with-handler からリターンする．この時の with-handler の返り値は *body* の返り値とする．*body* は第一引数で指定したすべての catch 節に動的に囲まれた式である．

body の実行中に例外が発生して，*body* の中で捕捉されなければ，with-handler の第一引数に指定した catch 節のリストに該当する catch 節があるか調べる．該当する catch 節があれば，例外を捕捉し，catch 節を除去して，対応する *handler* を捕捉した例外オブジェクトを引数に呼び出す．*handler* の呼出しの継続は with-handler の呼出しの継続とする．つまり，*handler* からのリターンは with-handler からのリターンとなる．*hendler* は同時に設置された（同じ with-handler 関数で設置された）どの catch 節にも動的に囲まれていないので，*handler* を呼び出す前に除去する catch 節は，例外を捕捉した catch 節と同時に設置されたすべての catch 節である．該当する catch 節がなかった時は，Java 言語と同様に，この with-handler 関数の呼出し式を動的に囲む catch 節をさらに検索する．catch 節は，Java 言語の try 文と Scheme 言語の with-handler 関数どちらにより設置されたかによらず，内側にある catch 節から順に検索される．

図 3.6 のプログラムは例外処理機能を使う例である．

この例で関数 f1 が呼び出されると，順に C1 クラスのメソッド m1，C2 クラスのメソッド m2，C1 クラスのメソッド m3 が呼び出される．m3 の throw 文は（1），(2)，(3）の三つの catch 節に動的に囲まれた式である．m3 の throw 文で，ExceptionX クラスの例外を投げると，まず，最も内側の（1）の catch 節がチェックされる．

C1.java:

```
public class C1 {
  public Object m1() {
    try { return m2(); }
    catch (ExceptionX e) {}              // (3)
  public Object m2() { return null(); }
  public Object m3() {
    try { throw new ExceptionX(); }
    catch (ExceptionY e) {}              // (1)
  }
}
```

Scheme:

```
(defclass C2 (C1))
(defmethod C2 m2 ()
  (with-handler
    (((class ExceptionX) (lambda (e) 'caught)))  ; (2)
    (lambda () (send this m3))))
(define (f1) (send (new C2) m1))
```

図 3.6: 例外処理機能を使う例

ExceptionX が ExceptionY のサブクラスでないとすると，この catch 節では捕捉されないので，次は（2）の catch 節がチェックされる．ここで例外オブジェクトのクラスが一致するので，捕捉されて例外ハンドラが起動される．その結果，'caught を返り値として，with-handler からリターンする．(3) の catch 節は検索されない．

例外の発生

Scheme クラスのメソッドの実行中にエラーが発生した際，例外を発生させて呼び出した側の関数にエラーを伝えることができれば便利である．そのため，Scheme 言語に throw 関数を追加する．これは以前のぶぶにあった例外処理機能で用いられていたのと同じ名前の手続きである．

(throw *exception*)

exception には Throwable クラスまたはそのサブクラス（Java クラスでも Scheme クラスでもよい）のインスタンスを指定する．throw 関数は，*exception* で表される例外を発生させ，この式を動的に囲む catch 節の検索を始める．

これらの機能により，例外をScheme言語で記述した例外ハンドラで処理する，Scheme言語で例外を発生させるといったことが可能になる．また，Java言語のメソッドからScheme言語のメソッドが呼び出され，さらにJava言語のメソッドが呼び出されている時に例外が発生すると，Java言語のメソッドの残りの計算だけでなく，Scheme言語のメソッドの残りの計算も打ち切って，呼び出した側のJava言語のメソッドの例外ハンドラに制御を移すといったことも可能になる．さらに，ここで追加した例外処理機能は，Scheme言語の他の言語仕様と矛盾しない．したがって，この拡張によりScheme言語の言語仕様に準拠しなくなることはない．

3.2.2 dynamic-wind関数の設計

dynamic-windは1998年に改訂されたScheme言語仕様[3]で初めて追加された．それ以前にもScheme言語用のライブラリの中にScheme言語で記述した実装はあったが，言語仕様として取り込まれてはいなかった．ぶぶでもScheme言語の言語仕様に沿ったdynamic-windを実現する．

まず，dynamic-wind関数の仕様を示す．

(dynamic-wind *before thunk after*)

before, *thunk*, *after*にはそれぞれ無引数の関数を与える．dynamic-windは*thunk*を実行し，その返り値をdynamic-wind自身の返り値とする．*thunk*の呼出しの動的寿命に入る時には必ず*before*が実行され，*thunk*の呼出しの動的寿命から出る時には必ず*after*が実行される．

関数の呼出しの**動的寿命**とは，関数が呼出されてから関数からリターンするまでの期間を指す．Scheme言語では，キャプチャした継続をいつでも呼び出すことができるので，*thunk*の呼出しの動的寿命は必ずしも連続しない．Scheme言語では，*thunk*の呼出しの動的寿命への出入りは次のタイミングで発生する．

- dynamic-wind関数により*thunk*が呼び出される時，*thunk*の呼出しの動的寿命に入る．
- *thunk*の呼出しの動的寿命の中で生成した継続を*thunk*の呼出しの動的寿命の外で呼び出した時，*thunk*の呼出しの動的寿命に入る．
- *thunk*からリターンする時，*thunk*の呼出しの動的寿命から出る．
- *thunk*の呼出しの動的寿命の外で生成した継続を*thunk*の呼出しの動的寿命の中で呼び出した時，*thunk*の呼出しの動的寿命から出る．

なお，継続呼出しを使って *before* や *after* の呼出しの動的寿命に入ろうとしたり，*before* や *after* の呼出しの動的寿命から出ようとした時の動作は定義されていない．

ぶぶでは，*thunk* の中で例外を発生させることによって，処理が *thunk* の呼出しから戻ることがある．そこで，関数呼出しの動的寿命から出るタイミングに以下の場合を追加する．

- ある関数呼出しの動的寿命の中で例外が発生して，発生した例外がその関数の動的寿命の中で設置されたどの `catch` 節にも捕捉されなかった時は，直ちにその関数呼出しの動的寿命を出る．

例外の発生により *thunk* の動的寿命を出るのは，`dynamic-wind` を動的に囲む `catch` 節を検索するよりも先である．したがって，*after* は `dynamic-wind` を動的に囲む `catch` 節の検索よりも先に行われることになる．

Scheme 言語の言語仕様では *before* と *after* の中で，それぞれ *before*，*after* の動的寿命の外でキャプチャした継続を呼び出した時の動作は未定義となっている．しかし，*before* や *after* の中でも例外が発生する可能性はある．この場合の動作が未定義のままでは問題である．ぶぶでは，*before*，*after* の中で行われるどのような継続呼出しや例外の発生も，継続を扱う機能や例外処理機能の仕様通りに動作するものとする．なお，*before*，*after* の実行中は *thunk* の呼出しの動的寿命にはまだ入っていないので，*thunk* の呼出しの動的寿命の外でキャプチャした継続の呼出しや，例外の発生において，それぞれ対応する *after*，*before* は実行されない．

この拡張により，*before* や *after* の実行中に継続の呼出しや例外の発生を行うことができるようになった．しかし，このままでは *before* や *after* の実行中に継続の呼出しや例外の発生を行った後，制御がどこに移るかが不明確である．これについては，Java 言語の finally 節の仕様に倣って，*before* や *after* の実行中に行われた継続の呼出しや例外の発生による制御を優先し，*before* や *after* に入る以前にあった行き先，返り値，および例外は破棄することにする．

ここで行った `dynamic-wind` に対する拡張は，Scheme 言語に対する拡張機能である例外処理機能への対応と，Scheme 言語で未定義となっていた部分を定義したにすぎない．したがって，拡張された仕様は Scheme 言語の言語仕様に矛盾することはない．

3.2.3 一級継続の設計

一級継続の障害

ぶぶへ一級継続を実装する際に問題となるのは，Java 言語ではプログラムから Java バーチャルマシンのスタックを操作できないという点である．ぶぶオブジェク

トシステムによってScheme言語で記述されたSchemeクラスのメソッドと，Java言語で記述されたJavaクラスのメソッドが相互に呼び出し合う．したがって，ぶぶのプログラムの継続は一般にJava言語で記述されたメソッドの実行も含んでいる．一級継続を実現するには，いつでもスタックを継続をキャプチャした時の状態に戻せる必要がある．しかし，Java言語ではJavaバーチャルマシンのスタックをプログラムから操作することができない．このような理由から，ぶぶに一級継続を完全に実現することは，一般には不可能である．

Javaバーチャルマシンに継続を扱うための命令を追加することでこれを解決することができる[8]．しかし，この方法はJava言語の持つポータビリティを犠牲にする．本論文では，ポータビリティを維持したまま，継続を扱う機能を実現する方法を検討する．

Java言語およびぶぶがマルチスレッドをサポートしているという点も問題となる．継続オブジェクトを，生成したスレッドとは別のスレッドで呼び出す時，どのようなに動作すれば良いかは自明ではない．それは，

- 継続がスレッドで実行されるべきか．
- 継続がなにを表すのか．

が決まっていないためである．これについては，文献[12]で議論されている．文献[12]では継続呼出しをCPS（Continuation Passing Style）風に拡張することでこの問題を解決している．つまり，継続呼出しにオプショナルな引数を追加して，継続を呼び出す時，呼び出された継続の実行が最後まで終わった後に呼び出す継続を陽に指定できるようにしている．継続を生成したスレッドとは別のスレッドで呼び出す時は，このオプショナル引数を指定する．これにより，継続の実行が終わると，継続を呼び出したスレッドの継続に制御を戻すことができる．

ぶぶ以外のJava言語で記述されたScheme言語処理系では，Schemeクラスを定義することができないので，Java言語で記述されたメソッドからScheme言語で記述された関数やメソッドが呼び出されることはない．また，Scheme言語レベルでサポートしない限り，マルチスレッドと継続を扱う機能を同時にサポートする必要もない．しかし，Scheme言語での関数呼出しをJava言語の関数呼出しで実現している処理系では，継続をキャプチャするためにJavaバーチャルマシンのスタックを操作しなければならないという同じ問題を持つ．

上で述べたように，完全な一級継続を実現することは，一般には不可能である．そこで，本研究では適当な制限のある継続を扱う機能を検討する．

制限を加えた一級継続

まず，いくつかの妥当な方式を考える．考えられる妥当な方式は大きく分けて次の三通りである．

方式1 下向きの継続呼出しだけ認める．

方式2 ぶぶの実行で最初に作られたセグメントでのみ継続の生成と呼出しを認める．

方式3 継続を生成したセグメント内でのみ呼出しを認める．

方式1はScheme言語の継続を扱う機能に，下向き呼出しに限るという制限を加える方式である．Java言語は下向きの継続呼出しに相当する例外処理機能を持っているので，実現は可能である．継続の呼出しを下向きに限れば，スタックの保存は必要なくなるので，継続の生成が軽くなるという利点がある．多くのJava言語で記述されたScheme言語処理系でもこの方式を採用している[20][21]．この方式の欠点は，例えScheme言語だけで記述されたプログラムであっても，上向きの継続呼出しが使えなくなる点である．そのため，既に動作しているプログラムをぶぶで実行することができない恐れがある．また，上向きの継続呼出しを用いることによってコルーチンやユーザレベルのマルチスレッドなど，様々な制御構造を簡単に記述できるというScheme言語の特徴を損なってしまう．さらに，この方式は継続を生成したスレッドとは別のスレッドで呼び出した時の問題は解決しない．

方式2では，キャプチャする継続にJava言語で記述されたメソッドの実行が含まれない．そのためSchemeスタックを持つぶぶでは実現は可能である．ぶぶオブジェクトシステムやマルチスレッドはScheme言語の言語仕様に対する拡張機能なので，これらの機能と継続を扱う機能は併用できなくてもScheme言語のみで記述されたプログラムを実行することができる．

方式3は方式2の制限を緩くした方式である．継続をキャプチャした後，そのセグメントからリターンするまでは，ぶぶの継続のうち，Java言語で記述された部分は実行されない．つまり，Javaスタック上のぶぶのプログラムが使っている領域は保存されている．そのため，スタックについてはSchemeスタックだけを保存しておけば良く，ぶぶで実現可能である．この方式では，継続の寿命が無限でないという点でScheme言語の継続ではないが，Scheme言語だけを使う範囲ではScheme言語の仕様に準拠した継続を扱う機能が利用できる．また，方式2と違って，Schemeクラスのメソッドの中で，Scheme言語で閉じたプログラムを書く限り，Scheme言語の仕様に準拠した継続を扱う機能が利用できる．つまり，Schemeクラスのメソッドから，Scheme言語で記述された既存の関数を呼び出すことがで

きる．この方式の欠点は，プログラマが継続オブジェクトの寿命が終わっているかどうかを判断しなければならない点である．

本研究では，

- Java 言語の持つポータビリティを維持する．
- Scheme 言語の仕様に準拠する．
- 方式 3 は方式 2 より記述能力が高い．

という理由により，方式 3 を選択する．

下向き継続呼出しの扱い

継続を生成したのと同じスレッドから呼び出す場合は，生成したのと別のセグメントからの呼出しであっても，下向きの呼出しであれば実現できる．

下向きの呼出しをする場合は，生成したのと別のセグメントからの呼出しであっても実現できる．これはセグメントからリターンするまでは Java スタックのうちぶぶのプログラムが使っている領域は保存されているからである．一方，対称性を考えれば下向きの継続であっても別セグメントからの呼出しであればエラーにするという方式も妥当である．本論文ではこの両方を実装する．

上向き継続呼出しの扱い

方式 3 において，呼び出す側と別のセグメントを実行中にキャプチャした継続を上向きに呼び出した時，単純にエラーにする以外にも，

- 可能な限り継続が呼び出されたように振る舞う．
- 継続オブジェクトの呼出しを本来の動作とは別の動作にする．

といった方法が考えられる．Java 言語で記述されたメソッドを含んだ継続の呼出しで問題となるのは，

A. 上向きの呼出しの際に，Java の例外ハンドラを設置できない．
B. 上向きの呼出しによって呼び出された継続の実行の中で，Java 言語で記述されたメソッドにリターンできない．

という点である．

B の問題は呼び出された継続がしばらく実行された後に起きる問題である．つまり，B の実行中問題が起きる前に別の継続を呼び出せば問題は起きない．そこ

で，呼び出された継続を，その継続が生成された時点から生成されたセグメントからリターンするまでの部分継続として扱う．これには，次のような方式が考えられる．

部分継続の実行後に例外が発生する方式 部分継続の呼出しの最後まで実行すると，部分継続の実行の最後に実行した式の値を持つ例外が発生する．

部分継続の実行後にリターンする方式 部分継続の最後まで実行すると，処理系がその部分継続の呼出しの継続を呼び出す．つまり，部分継続の呼出しからリターンする．この時の返り値は，部分継続の実行の最後に評価した式の値とする．

部分継続の実行後にリターンする方式は，文献[12]の方式と類似している．文献[12]における継続呼出しのオプショナルな引数に，暗にその部分継続の呼出しの継続を渡していると考えることができる．

Aの問題については，本来の継続の呼出しとは少し違う動作ではあるが，比較的妥当で実現可能な次の2つの方式が考えられる．

呼び出した側の継続を捨てる方式 呼び出された部分継続は，呼び出し側のセグメントで実行する．部分継続の呼び出しの（呼び出した側の）継続は，呼出しに伴って，呼び出したセグメントの残りの実行だけ捨てる．

呼び出した側の継続を残す方式 呼び出された部分継続は，呼び出した側のセグメントで実行する．呼び出したスレッドの継続は，呼び出した側の継続をすべて残したまま実行する．

「呼び出した側の継続を捨てる方式」と「呼び出した側の継続を残す方式」の動作を図3.7に示す．図3.7（1）は継続呼出しの前である．呼び出そうとしている継続は実行中のセグメントBとは別のセグメントで生成されたものであり，継続のCの部分だけが実行される部分継続として呼び出される．「呼び出した側の継続を捨てる方式」では，図3.7（2）のように，呼び出す側のBの部分を捨ててCを実行するが，「呼び出した側の継続を残す方式」では，図3.7（3）のように，Bを残したままCを実行する．

このように，下向きの継続呼出しの扱い以外に，上で述べた方式の組み合わせで4通りの方式が考えられる．

このうち，「呼び出した側の継続を捨てて部分継続を実行し，実行後にリターンする方式」には矛盾がある．

次に，「呼び出した側の継続を残す方式」は，「部分継続の実行後に例外が発生する方式」と組み合わせても，「部分継続の実行後にリターンする方式」と組み合わ

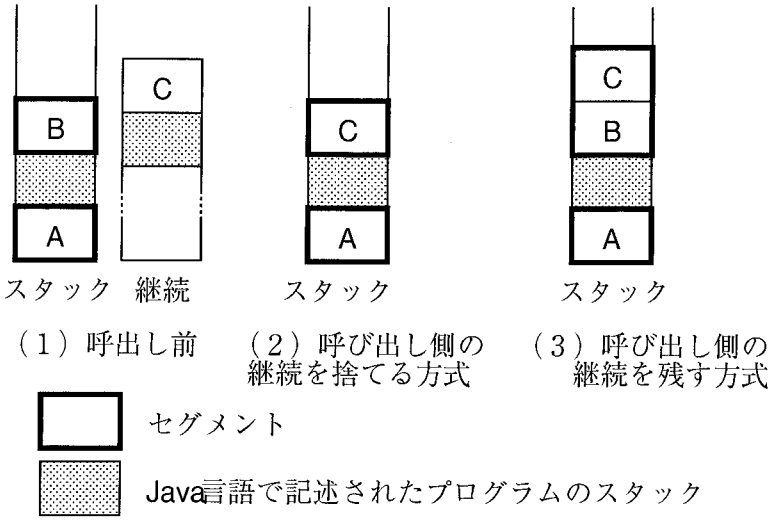

図 3.7: 部分継続の呼出しの概念図

```
(define (call-and-return cont arg)
  (with-handler (PartialContinuationException
                 (lambda(e) (slot-ref e lastValue)))
    (cont arg)))
```

図 3.8: 関数 call-and-return の定義

せても，同等の記述能力が得られることを示す．つまり，これらの組み合せは互いに他をシミュレートすることができる.「呼び出した側の継続を残して部分継続を実行し，部分継続の実行後に例外が発生する方式」で「呼び出した側の継続を残して部分継続を実行し，部分継続の実行後にリターンする方式」をシミュレートする場合は，例外ハンドラを設置した後に継続呼出しを行い，部分継続の実行の最後で発生する例外を捕捉すればよい．これを行う関数 call-and-return を図 3.8 に示す．ここで PartialContinuationException はセグメントからリターンする時に発生する例外で，その lastValue スロットに最後に評価した式の値が入っているものとしている．また，slot-ref はオブジェクトのスロットを参照する関数で，この例では e の lastValue スロットを参照している．さらに，図 3.9 に示すように call/cc を書き換えて，図 3.8 のようにして継続を呼び出すクロージャを生成するようにすれば，継続の呼出しの度に明示的に例外ハンドラを設置する必要もなくなる．

一方,「呼び出した側の継続を残して部分継続を実行し，部分継続の実行後にリターンする方式」で「呼び出した側の継続を残して部分継続を実行し，部分継続の実行後に例外が発生する方式」をシミュレートするには，まず継続を呼び出し

```
(define call-with-current-continuation
  (let ((old-cc call-with-current-continutaion))
    (lambda (proc)
      (old-cc
       (lambda (cont)
         (proc
          (lambda (arg)
            (with-handler (PartialContinuationException
                           (lambda(e) (slot-ref e lastValue)))
              (cont arg)))))))))
```

図 3.9: call/cc の再定義

```
(define (call-and-exception cont arg)
  (let ((val (cont arg)))
    (let ((e (new PartialContinuationException)))
      (slot-set! e lastValue val)
      (throw e))))
```

図 3.10: 関数 call-and-exception の定義

て，呼び出した継続から制御が戻ればその値を使って例外オブジェクトを作ればよい．これを行う関数 call-and-exception を図 3.10 示す．

ここで slot-set!はオブジェクトのスロットの値を書き換える関数で，この例では e の lastValue スロットを val の値で書き換えている．この場合も，call/cc を再定義することで継続呼出しの度に陽に例外を発生させる必要をなくすことができる．

本研究では,「呼び出した側の継続を捨てて部分継続を実行し，実行後に例外が発生する方式」と「呼び出した側の継続を残す方式」の 2 つの方式を実装する.「呼び出した側の継続を残す方式」については，制御に例外を用いない「部分継続の実行後にリターンする方式」との組み合わせを実装する．

ここで,「呼び出した側の継続を捨てて部分継続を実行し，実行後に例外が発生する方式」,「呼び出した側の継続を残して部分継続を実行し，実行後にリターンする方式」を新たにそれぞれ方式 A，方式 B と呼ぶことにする．実装手法を説明する前に，これらの方式での部分継続となる継続の呼出しと実行完了時の動作をまとめておく．

方式 A 部分継続の呼出しは次の手順で行う．

1. 図 3.11 (1) は呼出し前のスタックの状態である．

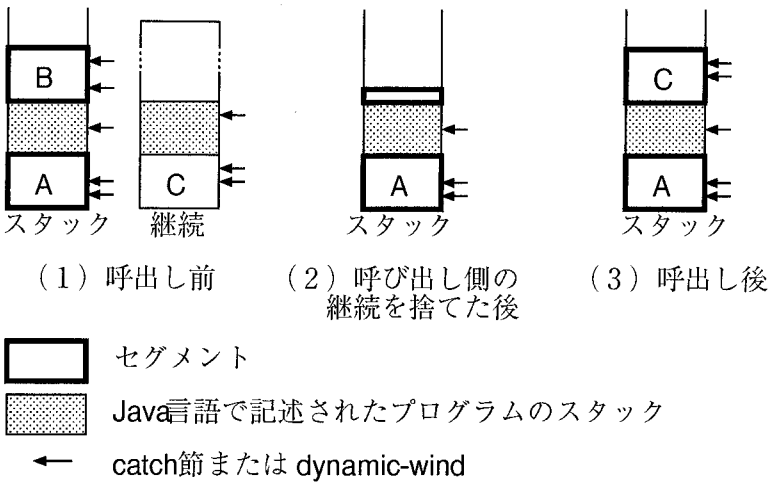

図 3.11: 方式 A の部分継続の呼出しの概念図

2. 部分継続の呼出しを動的に囲む `catch` 節の除去と，`dynamic-wind` による *after* の呼出しを適切な順序で行う．これが終わった状態が図 3.11（2）である．
3. 呼び出された部分継続を生成した式を動的に囲む `catch` 節のうち継続を生成したセグメントで設置されていた `catch` 節の設置と，`dynamic-wind` による *before* の呼出しを適切な順序で行う．これが終わった状態が図 3.11（3）である．
4. 呼び出された部分継続に制御を移す．

呼び出された部分継続を最後まで実行すると，例外が発生する．

方式 B 部分継続の呼出しは次の手順で行う．

1. 図 3.12（1）は呼出し前のスタックの状態である．
2. 呼び出された部分継続を生成した式を動的に囲む `catch` 節のうち継続を生成したセグメントで設置されていた `catch` 節の設置と，`dynamic-wind` による *before* の呼出しを適切な順序で行う．これが終わった状態が図 3.12（2）である．
3. 呼び出された部分継続に制御を移す．

呼び出された部分継続を最後まで実行すると，図 3.12（1）の状態になっているので，`catch` 節の設置，除去などは行わずに，最後に評価した式の値を返り値として，継続を呼び出した側に制御を戻す．

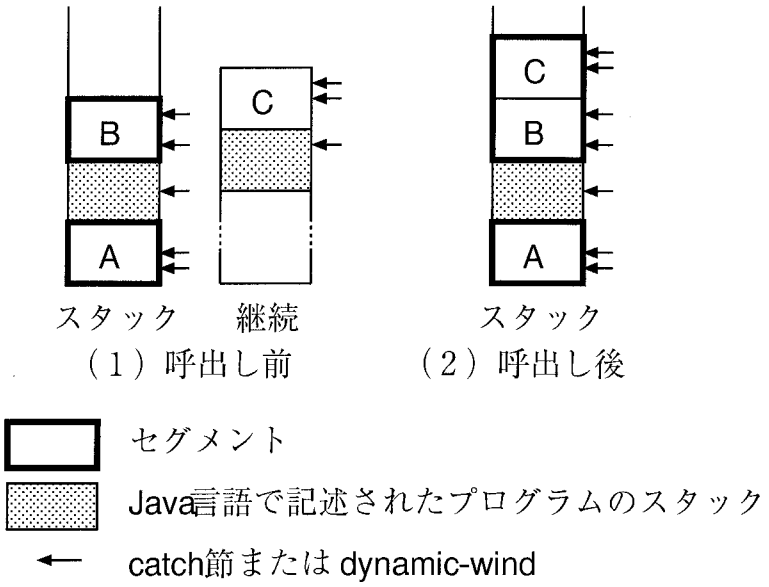

図 3.12: 方式Bの部分継続実の呼出しと行完了時の動作の概念図

別スレッドの継続の扱い

上向きの継続呼出しの扱いには，同じスレッドでなければならない点が含まれていない．したがって，継続をキャプチャしたスレッドと呼び出すスレッドが異なる場合も，上向きの継続呼出しと同様に扱うことができる．

3.3 実装

3.3.1 継続の実装

ぶぶは，Schemeスタック上にリロケータブルな関数フレームを生成する．また，リターンアドレスに相当する「リターン時処理」を書き換えてリターン時に特別な動作をする機構も既に備えている．インクリメンタル・スタック/ヒープ戦略は次の性質を持つ．

- 継続を使わないプログラムではオーバヘッドが発生しない．
- スタックの一度ヒープにコピーした部分を再度コピーしないため，`call/cc`が頻繁に呼び出される場合に効率良く働く．
- 実行にスタックを用い，スタック上の関数フレームはリロケータブルであることを前提としている．

- スタックアンダーフローを検出し，ヒープ表現になっている関数フレームをスタックに書き戻す．スタックアンダーフローは，関数のリターンアドレスの書き換えで効率良く検出できる．

以上の性質のため，本研究ではインクリメンタル・スタック/ヒープ戦略を採用する．インクリメンタル・スタック/ヒープ戦略については，2.4.3 節で説明した．ぶぶの実装では，スタックアンダーフロー時にスタックに書き戻す関数フレームのリストを持つレジスタを more レジスタと呼ぶ．

セグメントの境界

インクリメンタル・スタック/ヒープ戦略を使って 3.2.3 節で述べた方式を実装する際に，いくつか考慮する点がある．

継続をキャプチャしたのと同じセグメントから呼び出す場合，キャプチャしたセグメント以外の関数にはまだリターンしていない．したがって，継続をキャプチャしたセグメント以外の関数の関数フレームは，Scheme スタックおよび Java スタックに残しておくことができる．また，継続をキャプチャしたのとは別のセグメントから呼び出す場合，キャプチャしたセグメントの残りの実行だけを表す部分継続として扱う．したがって，継続オブジェクトはそれを生成した時のセグメントの関数フレームを保持しておけば十分である．そこで，継続をキャプチャする時には実行中のセグメントの関数フレームだけをヒープに退避することにし，それ以外はスタック上に残しておく．

これは次のようにして実現する．

- 仮のスタックの底を表すレジスタ（segment bottom レジスタと呼ぶことにする）を作り，セグメントが作られる時に，その時のスタックトップを segment bottom レジスタに設定する．

- 継続のキャプチャや継続の呼出し時のスタックの破棄などでスタックの底の位置が必要になる時は，segment bottom レジスタを参照する．

- セグメントが呼び出される時に古い more レジスタ，segment bottom レジスタをスタック上に退避し，リターン時に取り出す．

- セグメントの底の関数フレームの「リターン時処理」を書き換える．more レジスタに関数フレームが残っていれば，それをスタック上に書き戻して実行する．more レジスタに関数フレームが残っていなければ，本当にリターンする．これはスタックアンダーフローのルーチンに相当する．

継続呼出し時の判定

3.2.3節で述べた方式では，継続をキャプチャしたセグメントと呼び出すセグメントの関係で，呼び出した時の動作が変わる．方式によって，継続呼出しのタイミングで次の判定を行う．

- 継続をキャプチャしたセグメントと同じセグメントから呼び出される場合のみ完全な継続呼出しとする場合は，継続をキャプチャしたセグメントに入っているかどうかの判定．
- 下向き継続を常に完全な継続呼出しとする場合は，継続をキャプチャしたセグメントがスタック上に残っているかどうかの判定．

この判定には，セグメントを表すユニークなオブジェクト（segment IDと呼ぶことにする）を用いる．処理系は常に実行中のセグメントに対応するsegment IDを保持する．セグメントを作る時は，それまでのsegment IDをスタックに積み，新たにsegment IDを生成する．セグメントからリターンする時は，スタック上のsegment IDを取り出す．継続オブジェクトを生成する時は，その時のsegment IDを継続オブジェクトに記録する．継続を呼び出す時は，呼び出す側のsegment IDと継続オブジェクトが持つsegment IDを比較することにより，継続をキャプチャしたセグメントに入っているかどうかを判定することができる．

継続をキャプチャしたセグメントがスタック上に残っているかどうかは次のようにして判定する．まず継続をキャプチャしたセグメントに入っているかどうか判定して，継続をキャプチャしたセグメントに入っていれば，まだリターンしていないことが分かる．継続をキャプチャしたセグメントに入っていなければ，Schemeスタックのsegment bottomレジスタをたよりに，スタック上に退避されたsegment IDを探し，順に継続オブジェクトの持つsegment IDと比較する．継続オブジェクトが持つsegment IDがスタック上に見付かれば，継続をキャプチャしたセグメントはスタック上に残っていることが分かる．本当のスタックの底まで調べても一致しなければ継続をキャプチャしたセグメントは残っていない．

キャプチャしたセグメントからの継続呼出し

継続の呼出しがその継続をキャプチャしたセグメントと同じセグメントで行われていることが分かると，完全な継続呼出しを行う．継続をキャプチャしたセグメント以外の関数の関数フレームはSchemeスタックおよびJavaスタックに残っている．キャプチャしたセグメントの関数フレームは継続オブジェクトが保持しているので，これをmoreレジスタに設定して，Schemeスタックのsegment bottomレジスタの指す位置より上を破棄すればよい．

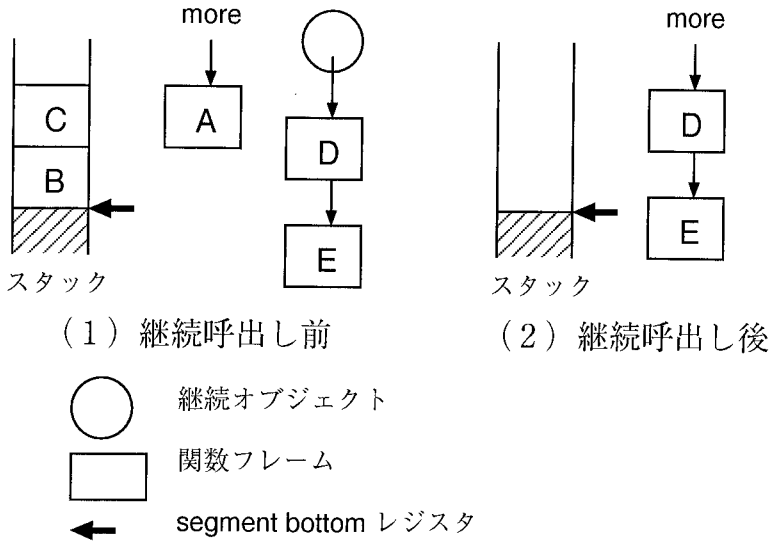

図 3.13: キャプチャしたセグメントからの継続呼出し

図 3.13（1）は継続を呼び出す前のスタックを表している．継続を呼び出すと，図 3.13（2）のように，スタック上の segment bottom レジスタの指す位置より上は破棄され，more レジスタに継続オブジェクトが持つ関数フレームのリストが設定される．

下向き継続の呼出し

継続をキャプチャしたセグメントと呼び出すセグメントが違っても，下向き呼出しであれば完全な継続呼出しを行う方式では，継続をキャプチャしたセグメントまで非局所的脱出を行った後，同一セグメントの継続呼出しと同様の方法で継続を呼び出す．非局所的脱出には Java 言語の例外処理機構を用いる．つまり，脱出先の segment ID を持つ例外を投げることにより非局所的脱出を行う．この例外のクラスはぶぶ処理系で予約し，ぶぶのプログラムでは使ってはいけないことにする．投げられた例外は各セグメントに対応するインタプリタで捕捉して，自分が脱出先であるかチェックする．インタプリタは自分が脱出先でなければ，捕捉した例外オブジェクトを投げ直す．

非局所的脱出に Java 言語の例外を用いるため，Java 言語の `catch` 節で捕捉されてしまう恐れがある．Java 言語の例外クラスはすべて `Throwable` のサブクラスであるため，`Throwable` を捕捉する `catch` 節ではぶぶがシステム予約として使っている例外でも捕捉してしまう．しかし，実用的な Java 言語のクラスライブラリがすべての例外を捕捉することはまれである．たとえ例外のトレースなどの目的ですべての例外を捕捉したとしても，必要な処理が終われば捕捉した例外オブジェ

クトを投げ直すので問題は起きない．また，脱出の際にJava言語の`finally`節が実行されるが，これは`dynamic-wind`の*after*が実行されるのと同様に正しい動作である．

方式Aにおける別セグメントの継続の呼出しと実行

3.2.3節で述べた方式Aでは，継続をキャプチャしたセグメントとは別のセグメントから呼び出した場合（下向きの継続呼出しでは常に完全な継続として扱う場合は下向きの継続呼出しを除く．以降も同様である．），呼び出した側の呼出しを行ったセグメントの残りの実行を捨てた後，呼び出された継続を部分継続として実行する．部分継続の実行が終わると例外が発生する．

この方式はsegment IDとは別に，実行中の継続が生成されたセグメントのsegment ID（continuation segment IDと呼ぶことにする）を処理系が保持することで実現する．continuation segment IDは，セグメントが作られた直後はそのセグメントのsegment IDになっている．継続の呼出しは，処理系の保持するcontinuation segment IDを呼び出される継続オブジェクトのsegment IDにすること以外は，キャプチャしたセグメントからの継続呼出しの場合と同じでよい．また，部分継続の実行は部分継続を呼び出したセグメントで行われているが，その部分継続を生成したセグメントのsegment IDはcontinuation segment IDとして保持されているので，継続をキャプチャする時はcontinuation segment IDを継続オブジェクトに記録する．

部分継続の最後まで実行した時に例外を発生させるために，スタックアンダーフローのルーチンで，moreスタックが空であることが分かった時部分継続の実行中かどうかを調べる．segment IDとcontinuation segment IDが一致しなければ部分継続の実行中である．部分継続の実行中ならJava言語で記述されたメソッドにリターンする代わりに例外を発生させればよい．

例として，まずセグメントAで継続Xを生成し（図3.14（1）），セグメントB以外でキャプチャした継続Yを部分継続として呼出し（図3.14（2）），さらに継続Yの実行中に継続Xを呼び出した（図3.14（3））場合を考える．この場合，継続Yの実行中に継続Xを呼び出すことで，セグメントAの継続の実行に戻っている．そのため，セグメントAからはセグメントAを呼び出したメソッドにリターンすることができる．

次の例として，まずセグメントAで継続Xを生成し（図3.15（1）），それをセグメントBで呼び出したとする．この呼出しは部分継続の呼出しとなるが，この実行中に継続Yを生成し（図3.15（2）），継続Yをさらにセグメントで呼び出したとする（図3.15（3））．継続Yはもともとセグメントで生成された継続X

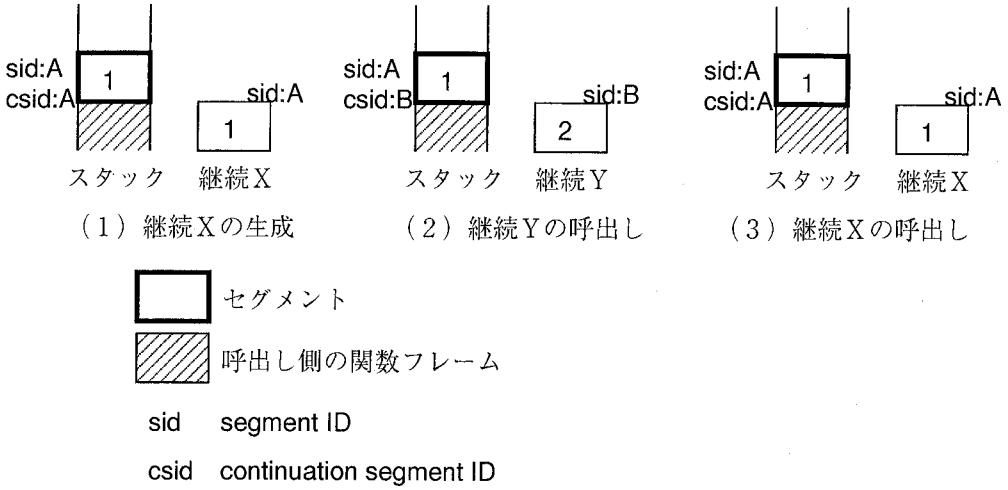

図 3.14: 方式 A の動作の例 1

の実行中に生成された継続であるので，セグメント A からはセグメント A を呼び出したメソッドにリターンすることができる．

方式 B における別セグメントの継続の呼出しと実行

3.2.3 節で述べた方式 B は，方式 A と違って，呼び出された継続の実行が本来セグメントからリターンする所まで進んだ時に，継続を呼び出した側に制御を戻す．そのため，ある継続が部分継続として呼び出されている時，呼び出した側の継続を保存しておく必要がある．部分継続として呼び出されている継続がその実行中に再び呼び出されることを考えると，呼び出した側の継続は LIFO に保存するのがよい．この LIFO を継続スタックと呼ぶことにする．

図 3.16（1）は継続のキャプチャ前の処理系の状態である．継続をキャプチャする時は，呼び出した側の継続のうち呼び出したセグメントの残りの実行を表す部分継続をキャプチャし，継続オブジェクトを生成して継続スタックに積む．この時，Scheme スタックおよび more レジスタは空にしておく（図 3.16（2））．この状態の継続スタックをコピーしたものを，Scheme 言語から見た時の継続オブジェクトとする．本論文では説明のために，特に断らない限り継続スタックに積まれるオブジェクトの方を指して継続オブジェクトと呼ぶ．関数フレームのリストやレジスタの値などは継続オブジェクトが保持するが，segment ID は継続スタックが保持する．継続スタックをリスト構造で実装すれば，継続をキャプチャするためにはリストの先頭の参照をコピーすれば良く，継続スタック全体をコピーする必要はない．`call/cc` の *proc* の呼出しからリターンする時にはスタックアンダーフローのルーチンが呼び出され，more レジスタから関数フレームを Scheme スタックに

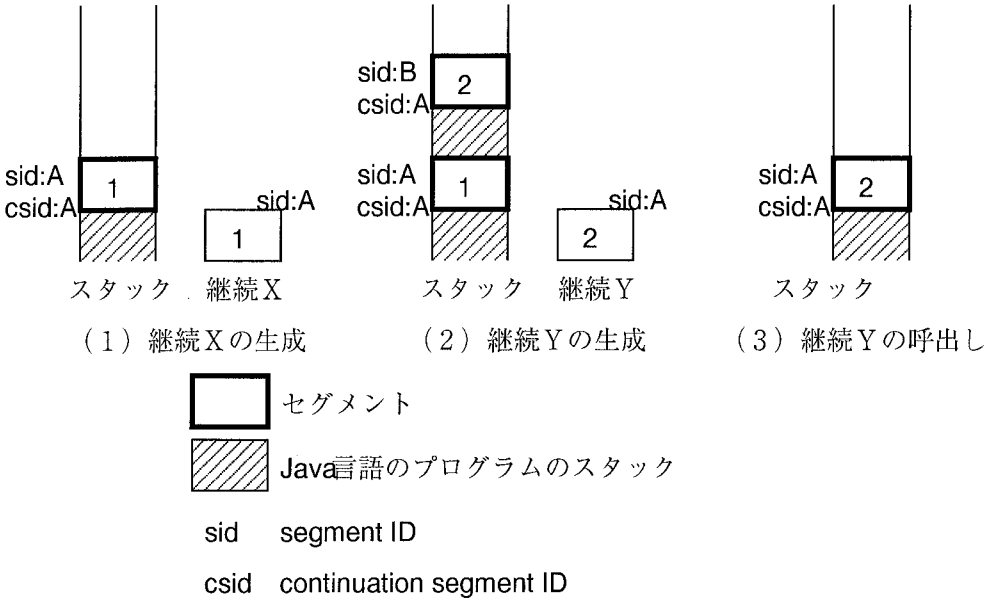

図 3.15: 方式 A の動作の例 2

戻そうとする．しかし，moreレジスタも空になっている．この方式では，moreレジスタが空の場合すぐにセグメントからリターンするのではなく，継続スタックに継続オブジェクトがある限り，これを取り出して実行する．継続オブジェクトを取り出した後の処理系の状態が図 3.16（3）である．このあと，さらに moreレジスタから関数フレームを Scheme スタックに戻し，moreレジスタから取り出した関数にリターンする．

図 3.17（1）は，継続が呼び出される前の処理系の状態および呼び出される継続である．継続が呼び出されると，継続スタックを次のように操作した後，Scheme スタックおよび more レジスタをクリアする．

- 継続をキャプチャしたセグメントと同じセグメントからの呼出しであれば，処理系の継続スタックを呼び出された継続オブジェクトの継続スタックで置き換える（図 3.17（2））．
- 継続をキャプチャしたセグメント以外のセグメントからの呼出しであれば，呼び出した側の部分継続を継続スタックに積んだ後，呼び出された継続オブジェクトが持つ継続スタックを継続スタックに追加する（図 3.17（3））．

方式 B における別セグメントの継続の呼出しと実行の改良

方式 B の，継続スタックを使う方式を改良することができる．方式 B では，継続の呼出しが部分継続の呼出しとなる時，呼び出した側の継続をキャプチャして継

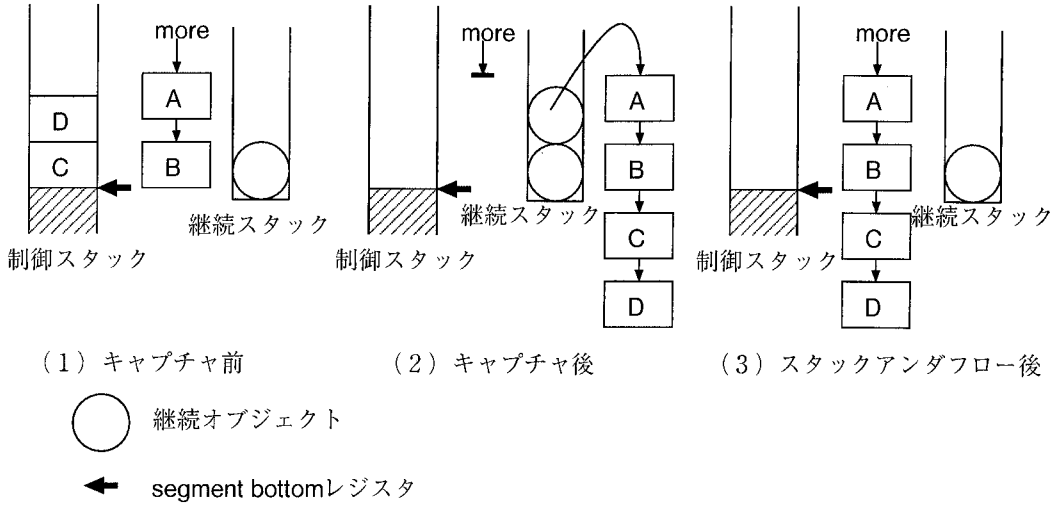

図 3.16: 継続スタックを使った継続のキャプチャ

続オブジェクトを生成している．これを普通の継続オブジェクトの生成と同様に行うと，継続の呼出しの際に，Scheme スタックをコピーすることになる．Scheme スタックのコピーはスタックの大きさに比例した，かなり重い処理である．しかし，呼び出された部分継続を実行中にさらに継続をキャプチャしないのであれば，部分継続を呼び出した側の継続は呼び出された部分継続からリターンした後に 1 回実行されるだけである．したがって，そのスタックをコピーする必要はない．そこで，部分継続を呼び出した時の呼び出した側の Scheme スタックをヒープにコピーする作業は，その実行中にさらに継続がキャプチャされるまで遅延する．

改良された継続スタックを使う方式の動作を説明する．まず，部分継続を呼び出す時に Scheme スタック上の関数フレームのヒープへの退避は行わない．継続スタックに積む継続オブジェクトには，退避された関数フレームのリストの代わりに，Scheme スタック上の退避するべき範囲を記録する．more レジスタに Scheme スタックの続きの関数フレームが入っている可能性があるが，部分継続を呼び出す時に more レジスタの内容（関数フレームのリストの先頭）をスタックに積むことで解決できる．Scheme スタック上の退避するべき範囲は部分継続からリターンするまで保存する必要があるので，部分継続の呼出しの時には，その時のスタックトップを新たなスタックの底に設定して，呼び出された継続の実行を開始する．このスタックの底は segment bottom レジスタとは別に current bottom レジスタで保持する．segment bottom レジスタはセグメントのリンクにも使われるからである．図 3.18 は図 3.17（3）を，ここで述べた手法を使って改良した図である．

部分継続の実行中に，さらに継続をキャプチャする時は，継続スタックの中を調べて，関数フレームのコピーが遅延されている継続オブジェクトがあれば，遅延されたコピーを行い完全な継続オブジェクトに変える．いったん完全な継続オ

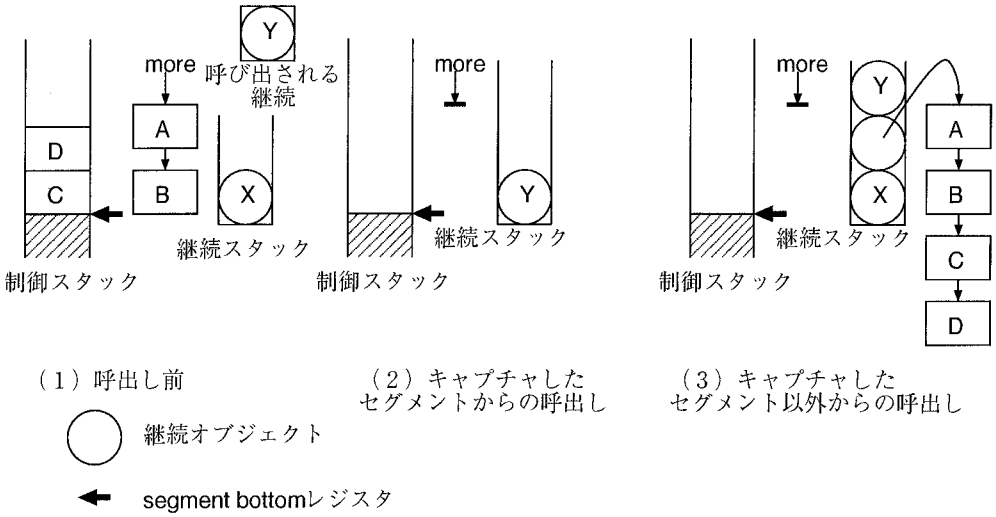

図 3.17: 継続スタックを使った継続の呼出し

ブジェクトになれば，それ以降の継続のキャプチャでは関数フレームのコピーが再び行われることはない．

3.3.2 例外処理の実装

例外処理機能には try 文に相当する with-handler 関数と，throw 文に相当する throw 関数がある．以下に 3.2.1 節で導入した構文を示す．

```
(with-handler ((class1 handler1)
               (class2 handler2)
               ...  )
  body)
(throw exception)
```

まず，with-handler 関数の実装について説明する．catch 節は with-handler の呼出しの動的寿命への出入りによって設置，除去される．そのため，catch 節は Scheme スタックと連動した LIFO で保持するのがよい．Scheme スタックに catch 節を保持することもできるが，本研究では Scheme スタックとは別に例外スタックを用意し，これをリスト構造で実装する．継続のキャプチャや呼出しなど，関数フレームのリストを操作する時は，例外スタックも同時に操作する．セグメントが呼び出される時の Scheme スタックへの退避なども，more レジスタの退避と同時に行う．

with-handler 関数が呼び出されると，第一引数に与えられた catch 節のリストに対応するオブジェクト（例外ハンドラ集合オブジェクトと呼ぶことにする）を

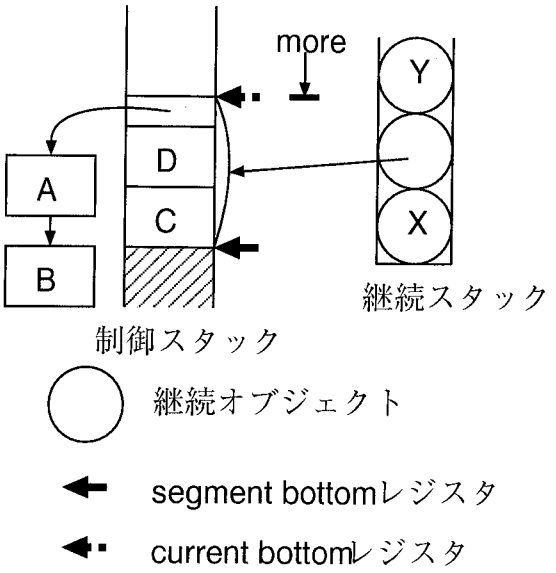

図 3.18: 改良した継続の呼出し

生成して，例外スタックに積む．catch 節を個別に例外スタックに積まないのは，例外を捕捉した時に，捕捉した catch 節を設置した with-handler により設置された他の catch 節も除去した後に例外ハンドラを呼び出すためである．例外ハンドラ集合オブジェクトは，catch 節以外に例外ハンドラ集合オブジェクトを生成した with-handler の呼出しからリターンする先の関数フレームも保持する．これは，例外ハンドラからリターンするためである．with-handler の呼出しからリターンする先は，この時点では Scheme スタック上にあるので，継続がキャプチャされるとヒープに退避される．そのため，継続のキャプチャ時に，例外スタックを調べて，スタックのインデックスで関数フレームを指していれば，それをヒープ上の関数フレームで置き換える．

図 3.19（1）は with-handler の呼出し前の処理系の状態である．with-handler は，例外ハンドラ集合オブジェクトを例外スタックに積むと,「リターン時処理」を書き換えて *body* を呼び出す（図 3.19（2））．*body* の呼出しから普通にリターンすると，書き換えられた「リターン時処理」の実行により，*body* の呼出しの前に積まれた例外集合オブジェクトを取り出して捨てる（図 3.19（1）に戻る）．

throw 関数は単に Java 言語の throw により例外を発生させるだけでよい．この例外は，Scheme インタプリタにより捕捉される．各セグメントに対応する Scheme インタプリタ（これは Java 言語で記述されている）は，すべての例外クラスのスーパークラスである Throwable を使ってあらゆる例外を捕捉し，次のように処理する．捕捉した例外がシステム予約の例外であれば，対応する処理を行う．システム予約の例外でなければ，この例外に対応する catch 節を例外スタックから探す．

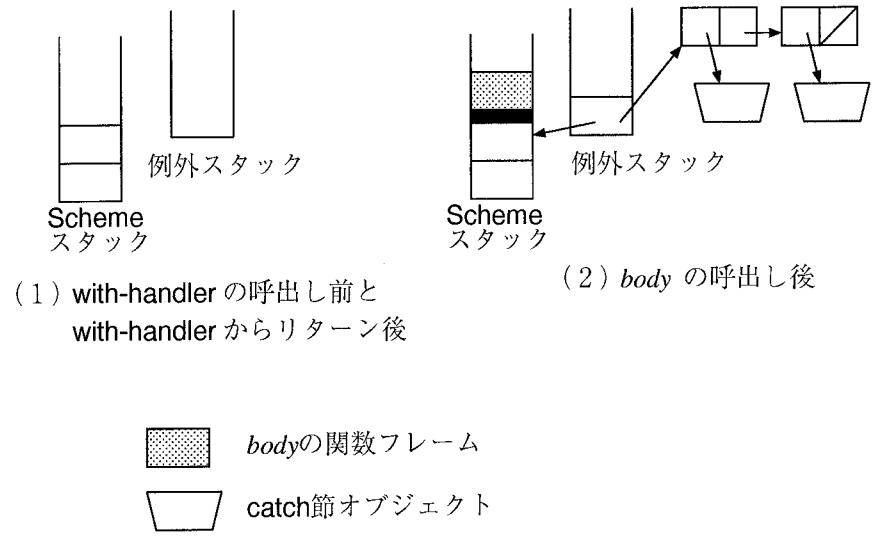

図 3.19: with-handler の呼出し

例外スタックに対応する catch 節が見付からなくても，方式Bであれば継続スタックから継続を取り出して catch 節の検索を続ける．catch 節が見付かれば，catch 節で指定された例外ハンドラを呼び出す．

例外を発生したセグメントでその例外を捕捉する catch 節が見付からない時は，Scheme インタプリタは捕捉した例外オブジェクトを投げ直す．投げ直された例外は Java 言語で記述されたメソッドの中で捕捉されるか，そうでなければ下のセグメントに到達する．これにより，Java 言語の catch 節と Scheme 言語の catch 節を正しい順序で検索することができる．

3.3.3 dynamic-wind 関数の実装

dynamic-wind はその *thunk* に与えられた関数の呼出しの動的寿命に入る時と出る時にそれぞれ *before*, *after* を実行する．

dynamic-wind を実現するには，*thunk* の呼出しの動的寿命と連動した LIFO を用いるのがよい．これは例外スタックと類似している．*thunk* の呼出しの動的寿命と連動した LIFO を dynamic wind スタックと呼ぶことにする．dynamic wind スタックにも例外スタックと同様に Scheme スタックを使ってもよいが，本研究では Scheme スタックとは別に，リストで実装する．継続オブジェクトは，関数フレームのリストや継続スタックとともに，生成された時の dynamic wind スタックも保持する．dynamic wind スタックは実行中の *thunk*（呼び出されていて，まだ呼出しの動的寿命から出ていない *thunk*）に対応した *before* と *after* の組（dynamic wind 組と呼ぶことにする）を *thunk* が呼び出された順に保持するスタックである．*thunk*

の呼出しの動的寿命に入る時は *before* を実行し，*before* 実行から普通にリターンすれば，dynamic wind 組を dynamic wind スタックに積む．また，*thunk* の呼出しの動的寿命から出る時は dynamic wind スタックから dynamic wind 組を取り出し，その *after* を呼び出す．このように，dynamic wind スタックの操作に伴って，*before* と *after* を呼び出せばよい．以下では，dynamic wind スタックを適切な状態に維持する手法について説明する．

thunk の呼出しの動的寿命に出入りが行われる可能性があるのは，次の時である．

- `dynamic-wind` が呼び出された時．
- 例外が投げられた時．
- 継続が呼び出された時．

`dynamic-wind` が呼び出された時　`dynamic-wind` が呼び出された時の動作は簡単である．`dynamic-wind` は，

1. *before* を呼び出す．
2. dynamic wind 組を生成し，dynamic wind スタックに積む．
3. *thunk* を呼び出し，その返り値を保存する．
4. dynamic wind 組を取り出し，*after* を呼び出す．

という一連の処理をする関数である．処理の途中に別の関数を呼び出すので，「リターン時処理」を用いて，処理を分割すればよい．この処理のために必要なデータの領域には，リターン時処理固有のデータの領域を使う．

例外が投げられた時　例外が投げられた時は，意味的には例外を捕捉した `catch` 節が設置された所まで関数呼出しを順に巻き戻して，例外ハンドラを呼び出す．巻き戻す関数呼出しの中に *thunk* の呼出しがあれば，dynamic wind 組を dynamic wind スタックから取り出して *after* を実行すればよい．

これには 2 つ問題がある．1 つ目の問題は，3.3.2 節で説明した実装では，例外を捕捉した `catch` 節が設置された所までの関数の巻き戻しを順に行っていない点である．3.3.2 節の実装では，例外ハンドラ集合オブジェクト（同時に設置された `catch` 節をまとめるオブジェクト）が `catch` 節を設置した `with-handler` 関数の呼出しからのリターン先の関数フレームを保持しており，リターン先関数フレームは中身をチェックせずに捨てている．この問題は，例外ハンドラ集合オブジェクトが，`catch` 節が設置された時の dynamic wind スタックを覚えておくことで解決できる．例外が投げられた時は，例外を捕捉する `catch` 節が設置された時の状態になるまで，dynamic wind スタックから dynamic wind 組を取り出して *after* を呼び

出す．実行中のセグメントに例外を捕捉する `catch` 節がなければ，dynamic wind スタックが空になるまで，dynamic wind 組を取り出して *after* を呼び出す．*after* を呼び出す時は，例外スタックを `dynamic-wind` が呼び出された時の状態に戻しておかなければならない．これは，例外スタックの先頭の例外ハンドラ集合オブジェクトが覚えている dynamic wind スタックを調べながら，余分な例外ハンドラ集合オブジェクトを例外スタックから捨てることで実現できる．

2つ目の問題は，*after* の呼出しからリターンした後に例外ハンドラの呼出しのための処理を続けなければならない．これは，*after* の「リターン時処理」を特殊なものにしておくことで解決できる．

継続が呼び出された時　まず，呼び出した継続が呼び出したのと同じセグメントでキャプチャされた継続だった場合について考える．継続オブジェクトはキャプチャされた時の dynamic wind スタックを保持している．方式Bでは，継続スタックに積まれた継続オブジェクト保持する dynamic wind スタックをつなぎ合わせたものが，呼び出した継続が持つ dynamic wind スタックであると考えればよい．また，処理系の持つ dynamic wind スタックも，継続スタックに積まれた継続オブジェクト保持する dynamic wind スタックを処理系の dynamic wind スタックの底につないだものと考える．継続が完全な継続として呼び出される際は，処理系の持つ dynamic wind スタックと呼び出される継続の持つ dynamic wind スタックを比較して，その差分を実行すればよい．手順は次のようになる．

1. 処理系の dynamic wind スタックだけが持つ dynamic wind 組がある限り，それを取り出して *after* を実行する．すべて実行し終わると次のステップに進む．
2. 継続オブジェクトの dynamic wind スタックだけが持つ dynamic wind 組はその *before* を呼び出して，処理系の dynamic wind スタックに積む．すべて実行し終ると，制御を呼び出した継続に移す．

次に，呼び出した継続が呼び出したセグメント以外でキャプチャされた継続だった場合を考える．この場合でも，方式Aでは継続が呼び出したのと同じセグメントでキャプチャされた継続だった場合と同じ動作でよい．方式Bでは，部分継続の呼出しの際に呼び出した側の継続はすべて保存される．呼び出された継続の継続スタックは，処理系の継続スタックに追加されるので，呼び出される継続の持つ dynamic wind スタックの *before* はすべて実行する．

この節で説明した実装手法では，*before* や *after* を実行中は，その呼出しからリターンした後にするべきこと，つまり *before* や *after* の呼出しの継続は，「リターン時処理」およびリターン時処理固有のデータとして完全に Scheme スタックに積まれている．したがって，*before* や *after* の呼出しの動的寿命の中でキャプチャした

継続，*before*や*after*の呼出しの動的寿命の外で呼び出しても問題は起きない．また，*before*や*after*の呼出しの動的寿命の外でキャプチャした継続を*before*や*after*の呼出しの動的寿命の中で呼び出した時も，問題となる点はない．

3.4 議論

3.4.1 オーバヘッド

3.3節で方式A，方式Bのそれぞれの実現方法を示した．3.3節で示した実装では，どの方式でも継続を扱う機能，例外処理機能，dynamic-windのどれも使っていない時は，余分なオーバヘッドがほとんど発生しない．発生する可能性のあるオーバヘッドをあえて挙げると，

- セグメント生成時における，各レジスタのスタックへの退避（方式により4～5個のオブジェクトの参照の配列への代入）と初期化およびセグメントからリターンする時の書き戻し，
- セグメントからリターンする時の継続スタックおよびmoreレジスタが空であることのチェック，
- セグメント開始時におけるsegment IDの生成

がある．これらのオーバヘッドがかかるのは，セグメント開始時とセグメントからリターンする時であり，いったんセグメントの実行が始まってしまうと，オーバヘッドは全くかからない．また，セグメント開始時とセグメントからリターンする時にかかるオーバヘッドも非常に小さく，無視できる．

3.4.2 実行効率の比較

次に方式Aと方式Bの実行効率の比較を行う．本研究で実装した制御機能を使わない時は，これらの方式の差は上に挙げたオーバヘッドの差しかない．

本研究で実装した制御機能を使う場合，方式Bは継続スタックを用いて方式Aよりも複雑な処理を行うので，方式Aよりも遅いと考えられる．方式Aでは方式Bと違って，呼び出す側の継続を保存しない．そのため方式Aでは，継続をキャプチャしたセグメント以外から呼び出す時にも，キャプチャしたのと同じセグメントから呼び出すのと同程度の時間しかかからない．一方，方式Bは継続を呼び出す側の継続を保存する．しかし，実装を改良をすることにより，継続の呼出しにかかる時間は方式A同様，通常の継続の呼出しと同程度の定数時間になる．継

表 3.1: 実装した方式の性能の比較（ミリ秒）

	tak	（比）	ctak	（比）	ctak/ep	（比）
方式A	2511	1.00	12399	1.00	22289	1.00
方式B	2503	1.00	13511	1.09	23293	1.05

続のキャプチャにかかる時間は，どちらの方式でも実行中のセグメントのSchemeスタックの長さに比例した時間がかかる．これは，インクリメンタル・スタック/ヒープ法の，継続のキャプチャにはスタックの長さに比例した時間がかかるという性質による．

実際に方式Aと方式Bの差を測定した結果を表3.1に示す．ここで測定に用いたtakは非常に小さな関数の呼出しとリターンを繰り返すプログラムで，本研究で実装した機能は使わない．ctakは，takで関数からのリターンを継続の下向き呼出しに置き換えたプログラムである．また，ctak/epは，takで関数からのリターンを例外の発生と捕捉で置き換えたプログラムである．表3.1によると，継続を使ったctakで1割程度の差がある．方式Bでは，生成した継続オブジェクトを，1回の関数呼出しにつき1回継続スタックに積み，2回（生成直後とリターン時）継続スタックから取り出す，合計3回の継続スタックへのアクセスが発生する．また，継続呼出し時にdynamic windスタックの管理の手間が増えるので，これによる効率の低下と考えられる．例外を使ったctak/epの効率の差も継続スタックを使ったことによる，処理の増加によるものと考えられる．

3.4.3 機能の比較

機能面で比較すると，方式Aと方式Bでは適用しやすいアプリケーションが異なる．

方式A 方式Aは，不完全ながら完全な継続に近い継続呼出しを実現する．方式Aでは，セグメント内で完結するタスクを別のセグメントに容易に移動させることができる．スレッドと組み合わせると，スレッドの間を移動するタスクを作ることができる．それぞれのスレッドでスケジューラを動かせ，スケジューラがタスクの継続を呼び出すような例を考える．これはScheme言語の無限の寿命を持つ一級継続を使った例としてよく見られる例を，マルチスレッドに拡張した例である．スケジューラはタスクの継続を呼び出す時に，呼び出す時の継続を明示的にキャプチャして保存する（またはタスクの継続の呼出しの引数に与える）．呼び出されたタスクの継続はある程度まで実行するとスケジューラの継続を呼び出す．その際にも，呼び出す時の継続を明示的にキャプチャして保存する（またはスケジュー

ラの継続の呼出しの引数に与える）．ここで保存した継続はどのスレッドで呼び出してもよい．これは方式Bではうまくいかない．方式Bでは部分継続の呼出しとなった継続呼出しの実行中の継続というのは，部分継続の最後まで実行した後に，呼び出した側の継続を実行するという意味を持つ．したがって，1つのタスクを複数のスレッドで実行すると，実行するスレッドが変わるたびに継続スタックが延びてしまい，無限にメモリを消費することになる．部分継続の呼出しが末尾呼出しとなる場合は，継続スタックは延びないが，これは非常に特殊な場合である．

方式B　方式Bは，完全な継続呼出しに近付けるというよりは，新たな機能を追加するという方式である．方式Bでは，別セグメントでキャプチャされた継続が実行が終わると，呼び出した側にリターンするので，部分継続となって消えてしまった部分の実行の代わりとなるプログラムを部分継続の呼出しからリターンした後に実行することができる．これはcatch節についても同じで，消えてしまったcatch節の代わりのcatch節を設置した状態で部分継続を実行することができる．例えば，入力ストリームと出力ストリームを受け取り，入力ストリームから読み出して，ある変換をして出力ストリームに書き込むフィルタのようなメソッドfilterを考える．filterメソッドは入力ストリームや出力ストリームの操作における例外を捕捉せずに，呼び出し側に任せる仕様である．Schemeクラスを定義してこのfilterメソッドををオーバライドしたとしよう．Schemeクラスのfilterメソッドでは，入力ストリームからブロックせずに読み出せるデータがなくなると，継続をキャプチャして，大域変数saved-contに保存してリターンする．しばらくして，保存してある継続を呼び出すが，この継続は実行中に入力ストリームや出力ストリームの操作における例外が発生する可能性がある．本来この例外は，filterメソッドを呼び出したJava言語で記述されたメソッドで捕捉されるはずであるが，saved-contに保存してある継続の呼出しは完全な継続の呼出しにはならないので，filterメソッドで発生した例外はJava言語で記述されたメソッドでは捕捉されない．方式Bでは次のようにすることで，Scheme言語で記述した関数の中で代わりに捕捉することができる．

```
(with-handler (IOException (lambda (e) (onIOException e)))
  (saved-cont #f))
```

この例では，filterメソッドの中で例外が発生した時にonIOException関数を呼び出す．方式Aと方式Bでは，saved-contに保存してある継続の呼出しの際にcatch節が除去されてしまうので，このような対応はできない．したがって，filterの中で発生した例外に正しく対応することが困難である．

このように，方式Aと方式Bは適しているアプリケーションが異なる．また，セグメントを越えた下向きの呼出しの扱いについても，どちらが良いとは言いが

たい．ところで，これらの方式のどの呼出しもサポートした処理系も考えることができる．本研究の方式では，継続の呼出しが完全な継続呼出しとなるかどうかは，プログラマが意識する必要がある．そこで，キャプチャされたセグメントからの呼出しまたは下向きの呼出しであれば，完全な継続呼出しとして動作し，それ以外はエラーとする．また，部分継続の呼出しを行う組み込み関数を方式Aと方式Bの2種類用意する．このようにすれば，アプリケーションによって，継続呼出しの方式を選ぶことができる．

3.4.4 応用

本研究で提案した方式は，Java言語上のScheme言語であるぶぶ以外にも応用できる．高級言語が低級言語で記述された関数を呼び出すネイティブプログラミングインタフェースを持つ場合，一級継続を実装することが難しいことが多い．しかし，本章で述べた手法を応用すると，完全ではないにせよ，様々な場面で有効に利用できる一級継続を実装することができる．

3.5 まとめ

本研究では，Java言語で記述されたメソッドとScheme言語で記述された関数が相互に呼び出すことのできるScheme言語の処理系ぶぶにおいて，可能な限り無限の寿命を持つ一級継続に近い機能を設計し，実装した．本研究で設計した方式はScheme言語の標準仕様に準拠しており，また，Java言語のメソッドを実行するJavaバーチャルマシンのスタックは直接操作しないため，Java言語の持つポータビリティは損なわない．

まず，Scheme言語で記述された関数を実行するためのスタックを，Java言語で記述されたメソッドの呼出しで区切って考える．この区切られたスタックの範囲をセグメントと呼ぶことにする．継続をキャプチャする時は，カレントセグメントのみをヒープに退避する．継続をキャプチャしたのと同じセグメントからの継続の呼出しは，Javaバーチャルマシンのスタック上に残っているJava言語の継続を利用して完全な継続として扱う．継続をキャプチャしたのとは別のセグメントからの継続の呼出しは，継続をキャプチャした時の，継続をキャプチャしたセグメントの残りの実行を表す部分継続の呼出しとして扱う．

継続を部分継続として扱う場合，1）呼び出した側の継続を残したまま実行するかどうか．2）部分継続の実行が終わった時に呼び出した側にリターンするかどうかという選択肢がある．本研究では，これらの方式のうち2つの方式の実装方法を示した．

本研究で設計した継続を扱う機能は，継続の一部をキャプチャすることができないような実装のインタプリタで，一級継続を実装する場合にも応用できる．

また，本研究ではJava言語の例外処理機能をシームレスに扱う機能と，Scheme言語の仕様に新しく追加された`dynamic-wind`関数を実装した．`dynamic-wind`関数はScheme言語の仕様を拡張して，部分継続の呼出しや例外処理機能と同時に使えるようにしてある．

本研究で実装した制御機能は，その機能を使わないプログラムには余分なオーバヘッドをかけない実装になっている．

第4章　遅延スタックコピー法

継続を呼び出すと，継続をキャプチャした関数に制御が移る．したがってその実行には，継続をキャプチャした関数の関数フレームが必要になる．また，継続をキャプチャした関数からリターンすると，その呼出し元の関数の関数フレームも必要になる．このように，継続の実行には，継続をキャプチャした時点でアクティブな関数フレームがすべて必要になる．一方，多くの処理系では関数呼出しとリターンの効率を良くするために，関数フレームをスタック上に確保している．したがって，リターンすると関数フレームは解放され，その領域は次に呼び出される関数の関数フレームのために再利用されてしまう．このような処理系では，継続をキャプチャする際に，スタック上の関数フレームをヒープにコピーすることで，将来の継続オブジェクトの呼出しに備える方式が一般的である．しかし，関数フレームはプログラムで多く使われるオブジェクトに比べ大きく，また，アクティブな関数フレームの数は関数呼出しの深さに応じていくらでも多くなるため，継続のキャプチャの処理は時間のかかる処理となっている．また，関数フレームのコピーを保持するための記憶領域も必要となる．

この章では，関数フレームをスタックに割り当てる処理系に効率のよい一級継続を実装する手法を提案する．提案手法である**遅延スタックコピー法**（lazy stack copying）は，2.4節で紹介した一級継続の実装戦略に相当する手法ではなく，関数フレームをスタックに割り当てる実装戦略を改良する手法で，このような実装戦略一般に利用できる．

この章では，まず4.1節で遅延スタックコピー法の基本的な動作を述べる．次に，遅延スタックコピー法を実装するに当たって必要となる機能の実装手法を4.2節と4.3節で述べる．4.2節では継続オブジェクトに呼び出される可能性が残っているかどうかを効率良く判定する手法を，4.3節では関数からのリターン時に特別な処理をするリターンバリアの実装手法を述べる．さらに，4.4節で遅延スタックコピーを実装した処理系を使った性能評価の結果を示す．4.5節では関連研究を示し，4.6節でまとめる．

なお，この章の内容は[22, 23]において発表した内容および，[24]において発表した内容の一部を含んでいる．

4.1 提案手法

遅延スタックコピー法は，プログラミング言語処理系に効率の良い一級継続を実装する手法である．本論文では，対象となるプログラミング言語としてScheme言語を想定しているが，一般のプログラミング言語にも当てはまる説明では高級言語と呼ぶことにする．遅延スタックコピー法は，一級継続を効率良く扱うために次の事実を利用している．

- 高級言語レベルで参照がなくなった継続オブジェクトは決して呼び出されない．
- スタック上にある関数フレームは，その関数にリターンしカレントフレームになるまで内容が保存される．また，非局所的な制御の移動を除けばその関数からリターンする前に解放されることもない．

call/ccは，call/cc自身の呼出しの継続をキャプチャする．キャプチャされた継続が無限の寿命を持つためには，その呼出し式を実行した時にアクティブである関数フレームに無限の寿命を与える必要がある．従来の一級継続の実装では，call/ccが呼び出されると，直ちにアクティブな関数フレームをコピーすることで，これを実現していた．しかし，スタック上の関数フレームは，非局所的な制御の移動を除けば，その関数からリターンするまで解放されることはない．そこで，遅延スタックコピー法では，継続のキャプチャ時にはスタックコピーを生成せず，代わりにスタック上の関数フレームを参照する継続オブジェクトを生成する．プログラムの実行が進み継続オブジェクトが参照している関数フレームからリターンすると，関数フレームは解放されてしまい継続を正しく呼び出せなくなってしまう．そこで，その関数からのリターン時，あるいは，それ以前に将来継続オブジェクトが呼び出される可能性が残っている場合に限り，関数フレームをヒープにコピーし，将来の継続の呼出しに備える．また，非局所的な制御の移動で継続オブジェクトが参照する関数フレームが解放される場合も同様にする．以下で詳細を述べる．

まず，プログラム全体の実行でcall/ccが一回だけ呼び出される場合について説明する．その後，一般の，call/ccが複数回呼び出される場合について説明する．

Scheme言語では，ある関数Fの実行中に

(call/cc *proc*)

式が評価されると，この式の継続がキャプチャされ*proc*が呼び出される．この様子を図4.1の（a）から（b）の遷移で示す．遅延スタックコピー法では，継続をキャプチャした時点では関数フレームのコピーを作らない．スタックのコピーの代わ

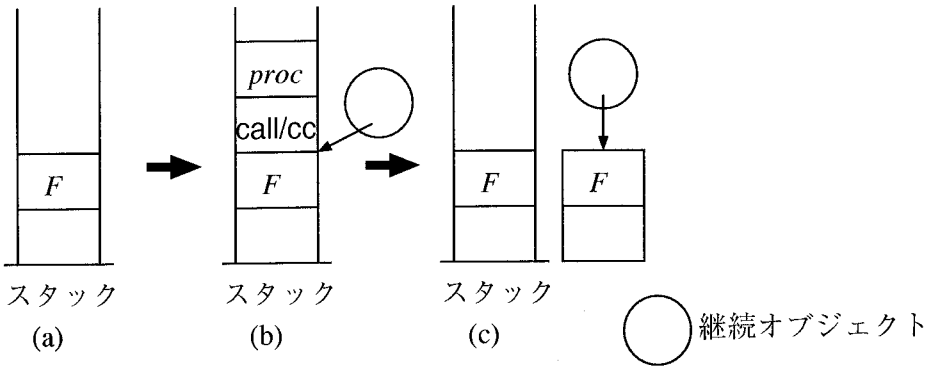

図 4.1: 遅延スタックコピー法での call/cc の呼出しとリターン

りに，その時のスタックトップのアドレスを持つ継続オブジェクトを作る．この継続オブジェクトが呼び出された時は，保存してあるスタックトップのアドレスを使って正しく call/cc 関数からリターンすることができる．この継続呼出しで必要な処理は，プログラムカウンタやスタックポインタの再設定だけである．

この後，実行が進み，call/cc を呼び出した関数 F からリターンすると，関数 F の関数フレームは解放されてしまう．そこで，関数 F からリターンする以前の適切な時点で関数 F の関数フレームのコピーを作る．また，さらに実行が進み，関数 F を呼び出した関数からリターンすると，その関数の関数フレームも解放されてしまう．そのため，継続がキャプチャされた時にアクティブであった関数フレームについて，その関数からリターンする以前の適切な時点で，ヒープにコピーする．ここでは，call/cc からのリターン時にすべてのアクティブな関数フレームをコピーすることとする．継続オブジェクトが関数フレームのコピーを持つようになることを，継続オブジェクトの**昇格**と呼ぶ．call/cc からリターンした時の様子を図 4.1 （c）に示す．

ところで，実際には，call/cc からのリターンは，*proc* からのリターンとなる．特定のリターンを捕捉し，特別な処理を行うことを文献 [25] に倣って**リターンバリア**（return barrier）と呼ぶことにする．リターンバリアの実装方法は 4.3 節で述べる．

遅延スタックコピー法の利点は，継続オブジェクトを昇格しようとした時，もしその継続オブジェクトがそれ以降決して呼び出されることがないことが分かれば，昇格させなくてもよい点である．これにより，不要な関数フレームのコピーが削減できる可能性がある．例えば，非局所脱出にのみ使われる継続オブジェクトは call/cc からリターンするまでの間にしか呼び出されることはない．したがって，call/cc からのリターンの時に継続オブジェクトを昇格させることにすると，非局所脱出にのみ使われる継続オブジェクトは昇格されなくなり，関数フレーム

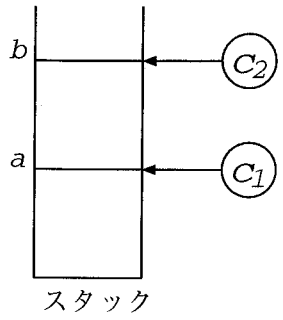

図 4.2: スタックと二つの継続オブジェクト

のコピーは作られない．継続オブジェクトが呼び出される可能性が残っているかどうかの判定には，スタックやレジスタ，大域変数など，高級言語のプログラムから直接参照される箇所に参照があるかどうかと，そこからポインタをたどることで到達できるかどうかで，保守的に近似することができる．これは，自動メモリ管理の手法であるごみ集め（garbage collection）において，あるオブジェクトが以降の実行で参照される可能性があるかどうかを判定するのと類似の近似である．高級言語のプログラムから到達できなくなった継続オブジェクトを，ごみ集めの用語を借りて**ごみ**（garbage）と呼ぶことにする．また，ごみ集めでは，スタックやレジスタ，大域変数など，高級言語のプログラムから直接参照される箇所を**ルート集合**（root set）と呼んでいる．継続オブジェクトがごみになったかどうかを効率良く見分ける方法については，4.2 節で述べる．

次に，一般の，`call/cc`が複数回呼び出され，複数の継続オブジェクトが生成される場合について補足する．継続オブジェクトが複数生成される場合，ある継続の呼出しによる大域的な制御の移動により，別の継続オブジェクトを昇格させる必要が生じる．図 4.2 は，`call/cc`が二回呼び出されて，継続 c_1 と，c_1 を含む継続 c_2 がキャプチャされた状態のスタックと継続オブジェクトの様子を模式的に示している．ここでは，どちらの継続をキャプチャした`call/cc`からもリターンしていないため，継続オブジェクトは両方ともスタックのコピーを持っていない．ここで，c_1 を呼び出すと，a と b の間にある関数フレームは解放されてしまうため，c_2 が呼び出される可能性がまだ残っていれば，c_2 を昇格させなければならない．一般に，ある継続 c を呼び出すと，c に含まれる継続以外でまだ呼び出される可能性の残っている未昇格の継続オブジェクトを昇格させなければならない．また，例外処理機構など，一級継続以外の制御機能を持つ処理系でも，大域的な制御の移動の際には，呼び出される可能性の残っている継続オブジェクトで，スタック上から消えてしまう関数フレームを参照しているものは昇格しなければならない．

このような継続オブジェクトを効率良く見付けるには，アクティブな継続オブジェクトをスタックを使って管理しておけばよい．このスタックを，**アクティブ継続スタック**（active continuation stack）と呼ぶことにする[1]．アクティブ継続スタックは，アクティブな継続オブジェクトを，生成された順に積んだスタックである．スタックの下位にある継続オブジェクトの継続は上位にある継続オブジェクトの継続に含まれる．処理系は，継続オブジェクトを生成すると，その継続オブジェクトをアクティブ継続スタックにプッシュする．call/ccからのリターン時には，そのcall/ccが生成した継続オブジェクトがアクティブ継続スタックのトップにあるので，それをポップし，必要であれば昇格させる．大域的な制御の移動の際は，アクティブな継続オブジェクトの中から，次の条件をすべて満たす継続オブジェクトを探し，昇格すればよい．

1. 呼び出される可能性が残っている．

2. 昇格されていない．

3. 参照しているスタック上の関数フレームが解放される．

さらに，アクティブ継続スタックを呼び出された継続がキャプチャされた時の状態に戻す．一方，アクティブでない継続オブジェクトは，条件1か条件2を満たさないため，昇格させる必要はない．これは，継続オブジェクトがアクティブでなくなる時（これはcall/ccからリターンする時である），もし，呼び出される可能性が残っていれば昇格されるためである．アクティブ継続スタックは，継続オブジェクトの単方向リストとして実装することもできる．なお，アクティブな継続オブジェクトの数は，高々アクティブな関数フレームの個数である．

アクティブ継続スタックを使うと，昇格する継続オブジェクトを効率良く見付けることができるだけでなく，継続オブジェクトの生成を減らすこともできる．関数の末尾呼出しにおいて，末尾呼出し式の継続が，呼び出した側の関数の呼出し式の継続と一致する点と，call/ccは引数に与えられた関数を末尾呼出しする点に注意すると，直前の継続のキャプチャから末尾呼出ししかしていない場合，末尾位置で継続がキャプチャされても新しく継続オブジェクトを生成する必要はない．キャプチャされる継続を表す継続オブジェクトは，アクティブ継続スタックのトップに積まれているはずである．直前に継続がキャプチャされてから，末尾呼出しのみにより呼び出された関数の実行中かどうかを効率良く判定する手法は，リターンバリアと密接に関連するため，4.3節で述べる．

[1]第3章の継続スタックとは別物であることに注意．

4.2 ごみの判定

遅延スタックコピー法では，継続オブジェクトを昇格しようとした時，将来その継続オブジェクトが呼び出される可能性が残っているかを調べる．4.1節では，継続オブジェクトが呼び出される可能性を，ルート集合からの到達性で近似することを述べた．ごみ集めに参照カウント（reference counting）GC[26]を用いている処理系であれば，継続オブジェクトの到達性は非常に小さなコストで調べることができる．このような処理系では，アクティブ継続スタックからの参照を考慮すると，call/ccからのリターン時に参照カウントが1である継続オブジェクトには到達性がないと言える．しかし，参照カウント法を使っている処理系は少なく，マーク・アンド・スイープ（mark and sweep）GC[26]やコピーGC[26]のようなトレーシング（tracing）によるごみ集めが実際には多く使われている．トレーシングによるごみ集めでも，call/ccからのリターン時に毎回ごみ集めを行えばルート集合からの到達性を正確に判定することができるが，これは本来関数からのリターンにかかる時間に対して，現実的な時間で実現することは難しい．そこで，この節では，さらに保守的な近似を用いて，ルート集合からの到達性を現実的な時間で調べる手法を述べる．ここでは，まず，変数のポインタを扱えないプログラミング言語を前提として説明し，その後に変数へのポインタを扱えるプログラミング言語について補足する．

まず，call/ccからのリターン時に，そのcall/ccが生成した継続オブジェクトがルート集合から到達できる可能性を列挙する．継続オブジェクトが次のいずれかの条件を満たす時，ルート集合から到達できる可能性がある．

1. 大域変数から直接参照されている．
2. ヒープにあるオブジェクトから参照されている．
3. call/ccからの返り値として返される．

条件2はルート集合から間接的に到達できる場合の考慮である．また，関数の返り値を格納するレジスタ以外で，関数からリターンした後もリターン前の値が参照される可能性のあるレジスタは大域変数と同様に考える．なお，リターンしようとしているcall/ccによって生成された継続オブジェクトが，スタック上の局所変数から直接参照されている可能性はない．これは次の理由による．変数のポインタなどを使わなければ，call/ccの呼出し元や，さらにその呼出し元の関数フレーム内にある局所変数に，call/ccからのリターンより前に書き込むことはできない．また，call/ccから呼び出された関数の関数フレームはcall/ccのリターンより前に解放されている．以上より，リターンしようとしているcall/ccが生成した継続オブジェクトへの参照を持つスタック上の局所変数は存在しない．

そこで，call/ccからのリターン時に，そのcall/ccが生成した継続オブジェクトがこれら三つの条件を満たすかどうかを調べればよい．しかし，条件1と条件2は，やはり現実的な時間で調べることは難しい．そこで，さらに保守的な次の条件を使う．

1'. 大域変数に代入されたことがある．

2'. ヒープにあるオブジェクトから参照されたことがある．

これらの保守的な条件は，継続オブジェクトが大域変数に代入されるか，ヒープにあるオブジェクトのスロットが継続オブジェクトで初期化された時，あるいは，ヒープにあるオブジェクトのスロットに継続オブジェクトが代入された時に，その継続オブジェクトにマークを付けることで効率的に調べることができる．このマークを**昇格予約フラグ**と呼ぶことにする．call/ccからのリターン時には，昇格予約フラグがセットされているか，または，その継続オブジェクトが返り値として返されている場合のみ，継続オブジェクトを昇格させる．

継続オブジェクトの大域変数やヒープ上のオブジェクトのスロットへの代入は，ごみ集めなどに使われる**ライトバリア**（write barrier）によって検出することができる．つまり，代入命令に，代入しようとしているオブジェクトが継続オブジェクトかどうかをチェックし，もしそうであれば昇格予約フラグをセットする処理を追加する．さらに，ヒープ上のオブジェクト生成時に，そのオブジェクトのスロットが継続オブジェクトへの参照で初期化される場合にも同様のチェックとマークが必要である．オブジェクト生成時に行う付加的な処理を**生成時バリア**と呼ぶことにする．

生成時バリアでは，関数クロージャと継続オブジェクトの生成に，特に注意が必要である．これらのオブジェクトは，生成時や昇格時に，スタック上にある局所変数を取り込むからである．以下で詳細を述べる．

関数クロージャ　関数クロージャ内では，クロージャを生成した時点で参照できる局所変数を参照することができる．つまり，クロージャは，生成した時点で参照できる局所変数の内容を持つヒープ上のオブジェクトである．したがって，クロージャの生成時には，局所変数に含まれている継続オブジェクトの昇格予約フラグをセットする必要がある．

例えば次のようなプログラムでは，(*) 印の行で生成されるクロージャは継続kを持つ．このクロージャを呼び出すことで，call/ccからリターンした後からでも，kを呼び出すことができる．

```
(call/cc
```

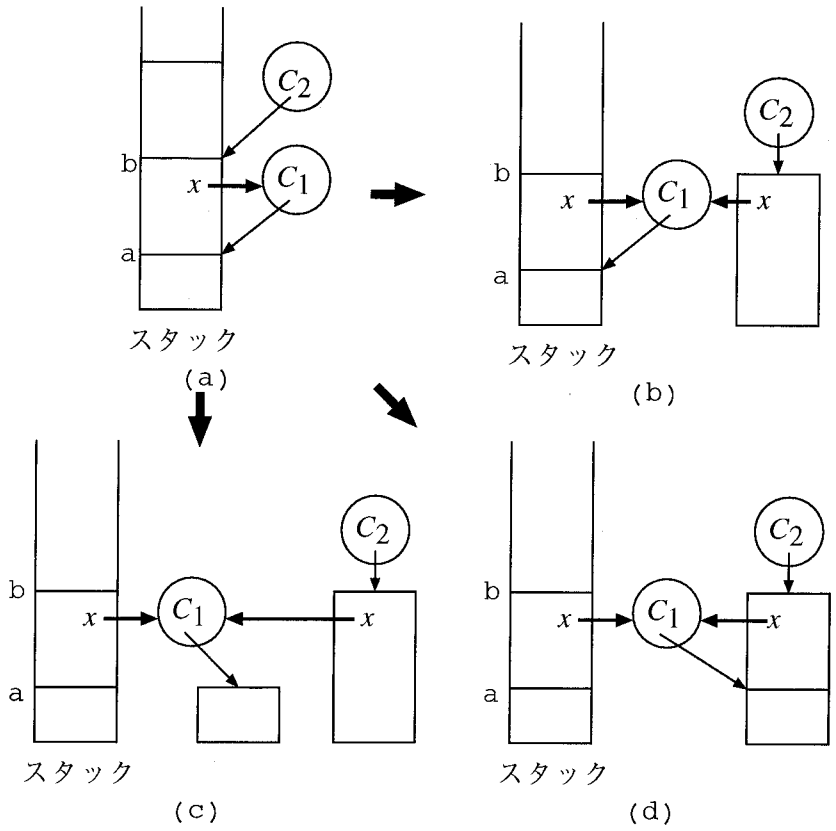

図 4.3: 継続オブジェクトの昇格による別の継続オブジェクトの昇格

```
(lambda (k)
  (lambda (v) (k v))))  ; (*)
```

しかし，必ずしもクロージャの生成時に参照可能なすべての変数の内容をチェックする必要があるわけではない．例えば次のようなプログラムでは，(*) 印の行で生成されるクロージャからは継続 k が参照できるが，実際には参照していないため，昇格予約フラグをセットしなくてよい．

```
(define (f) (call/cc
              (lambda (k)
                (lambda () 1))))  ; (*)
```

S式を実行前にコンパイルする処理系では，クロージャ内で参照されない変数はクロージャに含めないことが多い．このような処理系では，クロージャに含まれる変数のみをチェックすればよい．これにより，すべての局所変数をチェックする場合に比べて余分な昇格を減らすことができる．

継続オブジェクト　昇格した継続オブジェクトは関数フレームのコピーを持つ．もし，継続オブジェクトの持つ関数フレーム中の局所変数が別の継続オブジェクト

の参照を持てば，その継続オブジェクトは将来呼び出される可能性がある．図4.3の（a）と（b）は，昇格された継続オブジェクトが昇格されていない継続オブジェクトへの参照を持つようになる様子を示している．まず，継続c_1と，それを含む継続c_2がキャプチャされる．ここで，c_1もc_2もまだ昇格されていないとする．さらに，スタックボトムとbの間にある関数フレーム中の局所変数xがc_1への参照を持っているとする（図4.3の（a））．その後，継続オブジェクトc_2が昇格されスタックボトムからbの間の関数フレームがヒープにコピーされる．これにより，局所変数xもヒープにコピーされる（図4.3の（b））．もし，継続c_2が呼び出されると，高級言語は変数x経由でc_1を参照でき，c_1が呼び出される可能性が残る．一般に，ある継続cを持つ継続オブジェクトが昇格されると，cに含まれる継続はcが呼び出された後に呼び出される可能性がある．そこで，cに含まれる継続を持つ継続オブジェクトには昇格予約フラグをセットする．このような継続オブジェクトは，アクティブ継続スタックの，昇格される継続オブジェクトより下位にある継続オブジェクトである．

これにより正しく継続オブジェクトを昇格させることができるが，継続予約フラグをセットする代わりに，図4.3の（c）のように直ちに昇格してしまってもよい．さらに，複数の継続オブジェクトを同時に昇格する場合，スタック戦略の処理系でも，容易に図4.3の（d）のように複数の継続オブジェクトでスタックのコピーを共有できる．なお，スタックコピーの共有については，第5章でより効率の良い手法を述べる．

最後に，変数へのポインタを扱えるプログラミング言語で考慮する点について述べる．このような言語では，ポインタを介してカレントフレームより下位にある関数フレームの局所変数を更新することができる．そのため，call/ccの呼出し元の関数フレーム中の局所変数に，そのcall/ccが生成した継続オブジェクトへの参照が渡され，call/ccからリターンした後も継続オブジェクトが呼び出される可能性が残る．そこで，このような処理系では，ポインタを経由した書き込みもライトバリアにより監視し，昇格予約フラグをセットする．しかし，ポインタを経由した未昇格の継続オブジェクトの書き込みを検出しても，書き込み先のアドレスがスタック上で，かつ，書き込まれる継続オブジェクトの持つスタック上の関数ポインタのアドレスより上位であれば，昇格予約フラグはセットする必要がない．

4.3 リターンバリア

4.1節で述べた遅延スタックコピー法の実装では，call/ccからのリターン時に関数フレームのコピーを生成する．これには，call/ccからのリターンにバリア

(barrier) を張って，通常のリターンでは行わないスタックコピーの生成を必要に応じて行うようにする．末尾呼出しの際に，呼び出した側の関数フレームを残したまま，呼び出された側の関数を実行する処理系では，リターンバリアは容易に実装できる．このような処理系では，call/cc から呼び出された関数からリターンすると call/cc 関数の実行が再開するため，call/cc 関数の最後でスタックコピーの生成をすればよい．

しかし，真正に末尾再帰的である処理系では，末尾呼出しを次のいずれかの方法で実装するのが一般的である．

- 呼び出した側の関数フレームを呼び出された側の関数フレームとして再利用する．
- 一度呼び出した側の関数からリターンし，その上で，目的の関数を呼び出す．

このように末尾呼出しが実装された処理系では，呼び出された関数の実行が終わると呼び出した関数ではなく，さらにその呼出し元の関数に直接リターンすることになる．そのため，引数として渡された関数を末尾呼出しする call/cc 関数の最後にスタックコピーを生成するコードを追加しても，そのコードは実行されない．

また，4.1 節で，直前の継続がキャプチャされてから，末尾呼出しのみで呼び出された関数の末尾位置での継続のキャプチャでは，継続オブジェクトを生成するかわりに，アクティブ継続スタックのトップにある継続オブジェクトが利用できることを述べた．しかし，直前に継続がキャプチャされて以降，末尾呼出しのみで呼び出された関数の実行中であるかどうかは，リターンバリアと密接に関連するのでこの節にまわした．

この節では，遅延スタックコピー法のためのリターンバリアの実装手法としてリターン時検査，リターンアドレス置換，二重 call/cc 法の 3 つを述べる．これらの実装手法について，call/cc からのリターンにバリアを張る方法に加え，直前の継続のキャプチャから末尾呼出しのみで呼び出された関数の実行中かどうかを判定する方法を述べる．

4.3.1 リターン時検査

リターンバリアを単純に実装するには，関数からのリターン時に毎回スタックコピーが必要かを調べればよい．関数ごとに，その関数で継続をキャプチャしたかどうかを記録しておき，末尾呼出しではこの記録を呼び出される関数に引き継ぐ．関数からのリターン時にはこの記録を調べ，もし継続がキャプチャされていたならリターン前に適切な処理をする．

継続をキャプチャしたかどうかの記録は，末尾呼出しでない関数呼出しの直前で保存され，関数の実行の開始前に初期化されるフラグを用意すればよい．このフラグの値は，末尾呼出しでは初期化されない．call/ccは，継続オブジェクトを生成する前にフラグを調べる．もしフラグがセットされていれば，継続オブジェクトを生成する代わりにアクティブ継続オブジェクトのトップの継続オブジェクトを使う．もしフラグがセットされていなければ，継続オブジェクトを生成し，フラグをセットする．関数からのリターンでも，このフラグを調べ，もしセットされていれば，リターンの前にアクティブ継続スタックのトップの継続オブジェクトについて適切な処理をする．関数からのリターン後は，呼出し前のフラグの値に戻す．

さらに，スタック上の関数フレームの正確なアドレスが分かる場合には，より効率の良い実装が可能である．継続オブジェクトは生成された際に，スタック上の関数フレームのアドレスを持つ．継続オブジェクトが昇格するとこのアドレスは不要になる．しかし，昇格後もこのアドレスを保持しておくことで，前述のフラグなしでリターン時検査を実現できる．もし，アクティブ継続スタックのトップにある継続オブジェクトが，実行中の関数フレームからリターンする先の関数フレームを指しているなら，この継続オブジェクトは実行中の関数のリターン後の計算を持っている．つまり，前述のフラグがセットされた状態と見ることができる．継続オブジェクト持つ関数フレームのアドレスは，関数からのリターンのたびに参照されるが，アクティブ継続スタックのトップの継続オブジェクトが入れ替わる回数は十分少ないと期待できるため，このアドレスをキャッシュしておくと良い．

リターン時検査は，どのような処理系にも比較的簡単に実装できる．しかし，この実装は，call/ccを全く呼び出さないプログラムでも，リターン時の処理にオーバヘッドがかかり，効率のよい実装手法とは言えない．

4.3.2 リターンアドレス置換

リターンアドレス置換は，呼び出した側の関数フレームを呼び出された側の関数フレームとして再利用する処理系のための，効率のよいリターンバリアの実装手法である．関数フレームを再利用する処理系では，関数フレームに格納されているリターンアドレスは，末尾呼出しをしても保持される．そこで，call/ccの関数フレームのリターンアドレスを，リターンバリアの処理をするコードへのアドレスに置き換える．この時，本来のリターンアドレスは，置き換えの際に別のスタックに保存しておく．リターンバリアの処理のコードでは，必要な処理が終わると，別のスタックに保存された本来のリターンアドレスを使ってリターンす

る．call/ccが継続オブジェクトを生成する必要があるかどうかは，カレントフレームのリターンアドレスを調べれば良い．もし，リターンアドレスが既に置き換わっていれば，継続オブジェクトを新しく生成する必要はない．

この実装手法は，call/ccが呼び出されなければリターンバリアののない処理系とまったく同じ動作となるため，call/ccを呼び出さないプログラムには全くオーバヘッドがない．しかし，リターンアドレスが関数フレームのどこに格納されるかが処理系実装時に分かっていなければ，この手法を実装することはできない．C言語のように，関数フレームの構造を処理系実装時に決定できない言語で記述された処理系では，リターンアドレス置換によりリターンバリアを実装することはできない．

4.3.3 二重call/cc法

多くのインタプリタは，再帰的に定義された評価ルーチン（evaluator）で関数適用や組み込み関数のディスパッチを実装している．このような作りのインタプリタでは，call/ccから呼び出される関数専用の評価ルーチンを用意することで，関数フレームの構造に依存することなく，効率のよいリターンバリアが実現できる．call/ccから呼び出される関数専用の評価ルーチンでは，リターン時に必ずリターンバリアの処理をする．それ以外の関数を評価する評価ルーチンでは，リターンバリアの処理は一切行わない．この方式による実装はコンパイラにも応用することができる．それについては後述する．

二重call/cc法の擬似コードを図4.4に示す．図では，eval_callccがcall/ccから呼び出される関数専用の評価ルーチンで，evalがそれ以外の関数を評価する評価ルーチンである．evalがcall/ccを評価すると，新しい継続オブジェクトを生成し，継続スタックにプッシュする．その上で，call/ccの引数に与えられた関数を評価するために，eval_callccを呼び出す．eval_callccからリターンしようとすると，リターンバリアの処理が実行され，アクティブ継続スタックの継続オブジェクトが適切に処理される．一方，eval_callccは，call/ccを評価しても継続オブジェクトは生成せず，アクティブ継続スタックにある継続オブジェクトを使う．また，call/ccの引数に与えられた関数は，今度はeval_callcc 自身が評価する．二重call/cc法を使うと，末尾呼出しで関数フレームを再利用する処理系では，連続する末尾呼出しの中で一回目のcall/ccからの呼出しは通常の呼出しとなり，二回目以降のcall/ccからの呼出しでは関数フレームを再利用する呼び出しとなる．

この実装は，call/ccを呼び出さないプログラムには全くオーバヘッドがない．しかし，この方式によりリターンバリアを実装したインタプリタでは，最悪で一般

```
eval_callcc(rator, rands) {
  TOP:
    switch(type(rator)) {
    ...
    case CALL_CC:
        rator = first(rands);
        rands = make_rands(contstack_get_top());
        goto TOP; /* tail recursive call */
    ...
    }

    /* place the return barrier code here */

    return val;
}

eval(rator, rands) {
    switch(type(rator)) {
    ...
    case CALL_CC:
        cont = make_continuation();
        contstack_push(cont);
        if ((val = setjmp(cont->jmpbuf)) == 0)
            val = eval_callcc(first(rands), make_rands(cont));
        break;
    ...
    }
    return val;
}
```

図 4.4: 二重 call/cc 法の擬似コード

的なインタプリタのおよそ2倍までスタックが大きくなってしまう．これは，evalがcall/ccを評価するたびに，evalの関数フレームとほぼ同じサイズのeval_callccの関数フレームが余分にスタック上に確保されるためである．スタックが大きくなると，継続オブジェクトの昇格の際にスタックをコピーする量が増え，継続オブジェクトの昇格にかかるコストが大きくなる．この問題は，eval_callccをevalの中にインライン展開することで解決できる．

同様の手法がコンパイラにも応用できる．コンパイラの場合も，連続する末尾呼出し中の最初のcall/ccでのみ関数フレームを生成するようにする．連続する末尾呼出し中の最初のcall/ccかどうかは，カレントフレームのリターンアドレスにより調べられる．

4.4 性能評価

遅延スタックコピー法の性能評価のために，いくつかの処理系に実際に実装して，ベンチマークプログラムの実行時間を測定した．この節では，まず，使用するベンチマークプログラムを説明する．次に，スタック戦略のScheme言語のインタプリタであるSCM[14]と，インクリメンタル・スタック/ヒープ戦略のScheme言語に特化したバイトコードインタプリタであるTUTScheme[27]に遅延スタックコピー法の実装をした処理系での性能評価を示す．さらに，SCMと，スタック戦略のScheme言語に特化したバイトコードインタプリタであるMzScheme[15]について，スタック戦略でのスタックコピーの共有など，遅延スタックコピー法の洗練された実装をした処理系での性能評価を示す．

4.4.1 ベンチマークプログラム

ここでは，性能評価に使用するベンチマークプログラムを説明する．ここで使用するベンチマークプログラムのうち，tak，ctak，boyer，puzzleの四つのプログラムはGabrielベンチマーク[28]の一部である．またsame-fringeでは，[29]で示されているコルーチンマシンのコードを使っている．

tak

takは図4.5に示す関数を使って，自己再帰呼出しを繰り返す．tak関数自身の処理は非常に小さいため，実行のほとんどの時間を，関数呼出しとリターンに使うことになる．なお，関数呼出しのうち，1/4は末尾呼出しとなっている．この関数の特徴は，膨大な回数の再帰呼出しをするにもかかわらず，再帰呼出しの深さ

```
(define (tak x y z)
  (if (not (< y x))
      z
      (tak (tak (- x 1) y z)
           (tak (- y 1) z x)
           (tak (- z 1) x y))))
```

図 4.5: takプログラム

```
(define (ctak x y z)
  (call/cc (lambda (k) (ctak-aux k x y z))))

(define (ctak-aux k x y z)
  (cond ((not (< y x))
         (k z))
        (else (call/cc
               (ctak-aux k
                (call/cc (lambda (k) (ctak-aux k (- x 1) y z)))
                (call/cc (lambda (k) (ctak-aux k (- y 1) z x)))
                (call/cc (lambda (k) (ctak-aux k (- z 1) x y))))))))
```

図 4.6: ctakプログラム

はそれほど深くならない点である．そのため，スタックはそれほど大きくならない．また，call/ccは一切呼び出さない．

ctak

ctakはtakと同じ計算をする．しかし，takでは関数の実行が終わった時通常のリターンをするのに対して，ctakでは関数呼出しの際に継続をキャプチャしておき，それを呼び出すことによってリターンする．ctakのプログラムを図4.6に示す．ctak関数はインタフェースをtak関数に揃えるための関数で，ctak-aux関数が本体である．tak関数では自己再帰呼出しを4回行うが，ctak-auxでは，そのうち末尾呼出しではない3回でcall/ccにより継続をキャプチャして，継続オブジェクトを呼び出されるctak-auxに渡している．残る一回の末尾呼出しではダミーのcall/cc関数を置き，引数の計算の位置（ここは末尾位置とはならない）でctak-aux自身を呼び出している．この時，呼び出されるctak-auxには呼び出す側のctak-auxが受け取った継続オブジェクトを渡している．この継続オブジェクトを呼び出すことにより，複数のダミーのcall/ccの引数を計算中である関数を

飛び越えてリターンする．

このプログラムでも，takと同様にそれほど深い再帰呼出しはせずに，膨大な回数の自己再帰呼出しをする．さらに，ctakでは全体の75%の関数呼出しで継続がキャプチャされる．生成された継続オブジェクトは，必ず1回だけ下向きに呼び出される．このようにctakは継続が密集してキャプチャされ，かつ非局所脱出の用途に使われる例になっている．

ctakにはもう一点特筆すべき点がる．ctakではcall/ccの引数に関数クロージャを渡している．これらの関数クロージャが生成される位置から，継続オブジェクトへの参照を持つ局所変数が参照できる．しかし，この変数はクロージャ内では使われない．

boyer

boyerは項書換えによる定理証明プログラムである．boyerでは評価対象の式をリストで表現しており，非常に多くのセルオブジェクトを生成する．これらのセルオブジェクトの多くは，すぐにごみになる．また，計算の一部で単一化（unification）を行うが，その過程でオブジェクトのスロットへの代入も行う．call/ccは一切呼び出さない．

puzzle

puzzleは，ベクタで表現された盤面を書き変えながら，全探索によりパズルを解くプログラムである．ループと再帰呼出しを使って通常の全探索を行う．図4.7にpuzzleの中心である，trial関数を示す．この関数は，ループにより次の一手を試行錯誤する関数である．trialではループ式全体の継続をあらかじめキャプチャしておき，解が見つかった時だけ，この継続を呼び出してループから抜ける．puzzle全体でこの部分にのみ一級継続が使われている．実際には，ループ式全体が関数クロージャとなっているため，ここでの継続の呼び出しも非局所脱出の用途である．

same-fringe

図4.8と図4.9に示すsame-fringeは，一級継続を使ってコルーチンを実現したプログラムである．二つの木構造のデータを受け取って，それぞれの葉要素に格納された値を，木の左から右に順にスキャンした時，同じ値が同じ順で現れるかどうかを検査する．この計算をそれぞれの木を探索する二つのルーチンを交互に実行することで実装している．まず，片方の木Tを再帰呼出しにより探索し，左

```
(define (trial j)
  (let ((k 0))
    (call-with-current-continuation
     (lambda (return)
       (do ((i 0 (+ i 1)))
           ((> i typemax) (set! *kount* (+ *kount* 1)) ())
           (cond
            ((not
              (zero?
               (vector-ref *piececount* (vector-ref *class* i))))
             (cond
              ((fit i j)
               (set! k (place i j))
               (cond
                ((or (trial k) (zero? k))
                 ;(trial-output (+ i 1) (+ k 1))
                 (set! *kount* (+ *kount* 1))
                 (return #t))
                (else (puzzle-remove i j))))))))))))
```

図 4.7: puzzle の中心となる関数

端の葉を見付ける．ここで，継続をキャプチャしておき，他方の木 U を，やはり再帰呼出しにより探索し，左端の葉を見付ける．この葉の値が T の左端の葉の値と違っていれば，ここで検査は失敗する．同じ値を持っていれば，この時点での継続をキャプチャし，T の葉まで探索した継続を呼び出す．これにより，T の葉の探索が再開され，次の葉を探すことになる．次の葉が見つかると，また，継続をキャプチャし，U の葉まで探索した継続を呼び出す．このように，葉に到達するたびに他方の木の探索途中の継続を呼び出し，交互に探索を進めることで，同じ値が同じ順序で現れるかを検査する．ここでは，長さ 10000 のリストを木と見たてて，二つの同じ値を持つ木について検査した．

4.4.2 スタック戦略とインクリメンタル・スタック/ヒープ戦略の処理系

この節では，SCM と TUTScheme の二つの処理系に遅延スタックコピー法を実装し，ベンチマークプログラムを実行する．ここでは素朴な実装を行い比較する．スタックコピーの共有など，性能を向上させる手法を取り込んだ実装は 4.4.3 節で行う．

```
(define (make-coroutine f)
  (call/cc
    (lambda (maker)
      (let* ((LCS '*)
             (CC '*)
             (resume (lambda (dest val)
                       (call/cc
                         (lambda (k)
                           (set! LCS k)
                           (dest CC val)))))
             (detach (lambda (val)
                       (call/cc
                         (lambda (k)
                           (set! LCS k)
                           (CC val))))))
        (let ((x (call/cc
                   (lambda (k)
                     (set! LCS k)
                     (maker (lambda (cont val)
                              (set! CC cont)
                              (LCS val)))))))
          (detach ((f resume detach) x)))))))

(define (call dest val)
  (call/cc
    (lambda (k) (dest k val))))

(define (make-tree-walker tree)
  (make-coroutine
    (lambda (resume detach)
      (define (walk tree)
        (cond ((null? tree)
               #f)
              ((not (pair? tree))
               (detach tree))
              (else
                (walk (car tree))
                (walk (cdr tree)))))
      (walk tree))))
```

図 4.8: same-fringe のプログラム

```
(define (make-terminated-tree tree)
  (cons tree '*END-OF-TREE*))

(define (same-fringe tree1 tree2)
  (let ((c1 (make-tree-walker (make-terminated-tree tree1)))
        (c2 (make-tree-walker (make-terminated-tree tree2))))
    (let loop ((x (call c1 #f))
               (y (call c2 #f)))
      (cond ((not (eqv? x y))
             #f)
            ((eq? x '*END-OF-TREE*)
             #t)
            (else
              (loop (call c1 #f) (call c2 #f)))))))
```

図 4.9: same-fringe のプログラム（続き）

SCM はスタック戦略を使って一級継続を実装している Scheme 言語のインタプリタである．SCM は C 言語で記述されており，C 言語の制御スタックを Scheme 言語の制御スタックとして使っている．つまり，Scheme 言語での末尾位置以外の関数呼出しが C 言語の関数呼出しに対応している．SCM に遅延スタックコピー法を実装するにあたって，リターンバリアはリターン時に毎回検査するリターン時検査の方式（4.3.1 節参照）で実装した．関数クロージャの生成の際には，実行中の関数の局所変数を調べ，そこから参照される継続オブジェクトに昇格予約フラグをセットすることにした．これについて，さらに説明を加える．局所変数にはクロージャに閉じ込められた変数，関数呼出しにより渡された引数，および let 式等により導入された変数がある．このうち昇格予約フラグをセットする必要がある継続オブジェクトを持っている可能性があるのは，関数呼出しにより渡された引数のみである．SCM の関数呼出しでは，引数がリストとして呼び出される関数に渡されるため，ここでの実装では，関数呼出しのたびにこのリストを保存しておいて，関数クロージャの生成の際にこのリストを走査するようにした．

TUTScheme はインクリメンタル・スタック/ヒープ戦略を使って一級継続を実装している，Scheme 言語に特化したバイトコードのインタプリタである．そのため，ヒープにコピーした関数フレームは複数の継続オブジェクトで共有され，同じ関数フレームは一回しかコピーされない．TUTScheme では，関数からリターンするたびにリターン先の関数フレームだけをスタックからヒープにコピーする実装も可能だが，今回は継続の昇格の際にスタック全体を一括してコピーするようにした．リターンバリアは，TUTScheme が関数のリターンアドレスを容易に書き

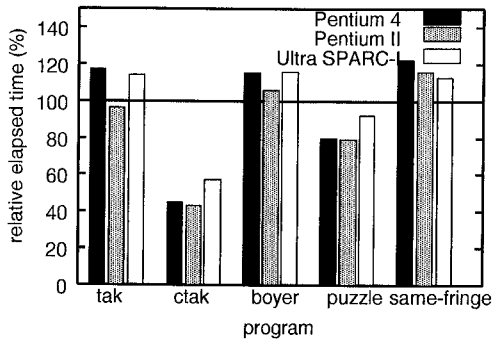

図 4.10: SCM によるベンチマークプログラムの実行時間

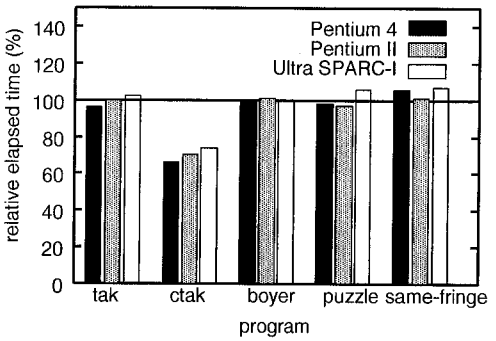

図 4.11: TUTScheme によるベンチマークプログラムの実行時間

替えることができるため，リターンアドレス置換（4.3.2 節参照）で実現した．また，TUTScheme は関数クロージャから参照される局所変数をヒープに移すバイトコードを持つため，このバイトコードを見張ることで関数クロージャから参照される継続オブジェクトの昇格予約フラグをセットすることにした．TUTScheme では，関数クロージャから実際に参照される可能性がある変数しかヒープに移さないため，関数クロージャ内から参照される可能性のない継続オブジェクトが昇格されることはない．

このように実装した二つの処理系でのベンチマークプログラムの実行時間を図 4.10 と図 4.11 に示す．実行には，Pentium 4，Pentium II，Ultra SPARC-I のそれぞれのプロセッサを使った．いずれも遅延スタックコピー法を実装していない処理系で同じプログラムを実行した時の時間を 100%として，実行時間の比を示している．

ベンチマークプログラムの実行時間を見ると，SCM，TUTScheme とも，`ctak` では実行時間が大幅に短くなっている．特に SCM では実行時間の減り方が大きい．`ctak` では非常に多くの非局所脱出のための継続オブジェクトが生成されるが，それらの多くがヒープ上のオブジェクトから参照されることなくごみになる．そのため遅延スタックコピー法がうまく働き，これらの多くの継続オブジェクトの

昇格を抑制したと考えられる．SCMの方が遅延スタックコピー法により実行時間の減り方が大きくなった理由は，もとの処理系の効率の差によると考えられる．TUTSchemeではインクリメンタル・スタック/ヒープ戦略により，遅延スタックコピー法を使わない場合でも同じ関数フレームを二回以上コピーすることは防げていた．しかし，スタック戦略で一級継続を実現するSCMでは，遅延スタックコピー法を使わない場合，継続オブジェクトの生成のたびにスタック上の関数フレームをコピーしていた．そのため，遅延スタックコピー法なしでは非常に効率が悪かった．SCMとTUTSchemeでは実装方式が様々な点で異なるため単純に比較はできないが，遅延スタックコピー法を実装していない処理系同士では，`ctak`の実行に要した時間はTUTSchemeの方が圧倒的に短かかった．

`puzzle`では，SCMでは`ctak`ほどではないが，実行時間が短くなっている．しかし，TUTSchemeでは，ほとんど改善が見られない．これも，インクリメンタル・スタック/ヒープ戦略で一級継続を実装しているTUTSchemeでは，もともと継続オブジェクトの生成にそれほど時間がかからず，また，関数フレームのコピーが作られる数も少ないため記憶領域を消費することも少なかったためと考えられる．

次に，`call/cc`を一切呼び出していないプログラムのSCMでの実行時間が大幅に遅くなっていることが目につく．この主な原因は，リターンバリア，ライトバリア，および生成時バリアである．しかし，ライトバリアと生成時バリアはTUTSchemeでもクロージャの生成時バリアを除けば同様の実装になっているため，このオーバヘッドの多くはリターンバリアによるものと考えられる．SCMのリターンバリアは，リターン時検査で実装した．このオーバヘッドは非常に大きく，Pentium IIによる結果を除けば20%近くになっている．Pentium IIで実行時間が極端に短い理由は今のところよく分かっていない．

`same-fringe`では，すべての継続オブジェクトが最終的に昇格されるため，遅延スタックコピー法による実行速度の改善は全く望めず，オーバヘッドのみ現れるはずである．実際，SCMとTUTSchemeの両方の処理系で，遅延スタックコピー法を実装していない処理系より遅くなっている．SCMとTUTSchemeで遅くなりかたに違いがあるのも，やはり，リターンバリアの実装方法によるものと考えられる．

以上のように，スタック戦略とインクリメンタル・スタック/ヒープ戦略のどちらの戦略で一級継続を実装した処理系でも，遅延スタックコピー法により一級継続を使ったプログラムの実行効率を高めることができた．特に，一級継続がそれほど効率良く扱うことができないスタック戦略の処理系では，性能の向上が顕著であった．しかし，リターンバリアの実装方法によっては一級継続を使わないプログラムにも無視できないオーバヘッドがかかることが分かった．

4.4.3 スタック戦略の処理系での洗練された実装

ここでは，スタック戦略により一級継続を実装する処理系に遅延スタックコピー法を実装し，性能を評価する．その際，いくつかの性能を改善する手法を取り込む．

まず，4.4.2節で使ったSCMに二重call/cc法によるリターンバリアを実装し，リターンバリアのオーバヘッドを削減する．また，スタックコピーの共有の手法を実装した処理系も作り，その効果を調べる．

さらに，別のScheme言語の処理系であるMzScheme[15]に同様の方式を使って遅延スタックコピー法を実装し性能測定した結果も示す[2]．MzSchemeはScheme言語を専用のバイトコードにコンパイルし，それを実行するインタプリタである．SCM同様C言語で記述されておりC言語の制御スタックをバイトコードの制御スタックとして使っている．また，一級継続も，スタック戦略により実装している．MzSchemeとSCMとの重要な違いは，MzSchemeでは関数クロージャ生成時に，クロージャ内から参照されないオブジェクトをクロージャに取り込まないようにしている点である．したがって，MzSchemeではクロージャ内で参照されない継続オブジェクトは，クロージャ生成時に参照できる局所変数に入っていても，昇格されることはない．

洗練された遅延スタックコピー法の実装によるベンチマークプログラムの実行時間を図4.12に示す．性能評価に用いたプロセッサはPentium 4とUltra SPARC-IIeである．図では，“original”が従来の処理系での実行時間，“lazy”が継続キャプチャ時のスタックのコピーを遅延させた処理系での実行時間，“share”がスタックのコピーの遅延に加えて，スタックのコピーの共有も行う処理系での実行時間である．また，ここでは実行時間の内訳のうちごみ集めにかかった時間を計測し，“GC”で表している．

遅延スタックコピー法の実行結果には，4.4.2節と似た傾向が見られるが，4.4.2節のSCMの実行結果と比べて，takでのオーバヘッドが小さくなっている点が異なる．MzSchemeでもtakのオーバヘッドは同程度である．これは，二重call/cc法によるリターンバリアの実装により，リターンにかかるオーバヘッドが小さくなったためと考えられる．しかし，オーバヘッドが完全には解消されていないことも分かる．これは，ライトバリアや生成時バリアによるものと考えられる．

次に，スタックのコピーを共有することによる効果は，SCMのctakでのみ，顕著に現れている．これは，SCMでは，遅延スタックコピー法を使っても，関数クロージャの生成により継続オブジェクトを昇格してしまっていたところを，スタックのコピーを共有することで，コピーの量を少なくすることができたためと考えられる．MzSchemeでは，遅延スタックコピー法だけを実装した処理系でも関数

[2]MzSchemeでの実装は皆川宜久氏によるものである．

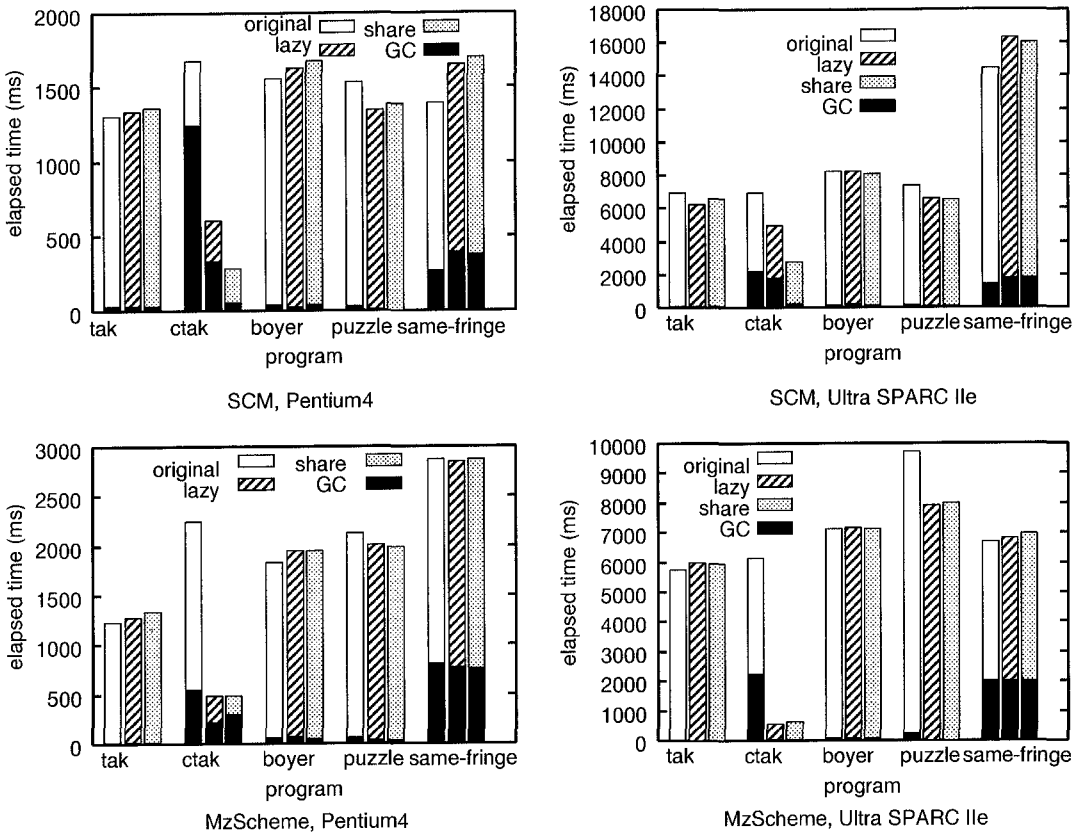

図 4.12: 洗練された実装による処理系での実行時間

フレームを一切コピーしなかったため，スタックコピーの共有によりさらに速度が向上することはなかった．また，puzzle では SCM でも MzScheme でも，遅延スタックコピー法のみで関数フレームを一切コピーしなくなっていたため，スタックコピーの共有による効果はなかったと考えられる．

また，ctak や same-fringe では実行時間の多くをごみ集めに費やしている．これが，ctak では遅延スタックコピー法により，大幅に削減されている．このことから，遅延スタックコピー法による高速化には，関数フレームのコピーにより記憶領域が占有されるのを抑制したことによる貢献も大きいことが分かる．

ところで，コルーチンのプログラムで，それぞれのコルーチンが脱出のために継続を使うことがある．脱出のための継続オブジェクトは，スタックのコピーの遅延により昇格される前にごみになる．しかし，継続オブジェクトがごみになる前に別のコルーチンに制御を移すと，その時に昇格されてしまうため，スタックコピーの遅延の効果はない．スタックのコピーの共有を行うと，コルーチンの制御のための継続オブジェクトが持つスタックのコピーと，脱出のための継続オブジェクトが持つスタックのコピーが共有される．実際，same-fringe の各コルーチンが行うリターンを継続を使った脱出に置きかえたプログラムで（問題の規模

は小さくしてある），Pentium 4 の環境で SCM を使って実行時間を計測したところ，遅延スタックコピー法を実装していない処理系で 960 ミリ秒，スタックのコピーの遅延を行う処理系で 1150 ミリ秒，スタックのコピーの共有を行う処理系で 810 ミリ秒であった．

以上のように，二重 call/cc 法によりリターンバリアを実装することでオーバヘッドを，かなり小さくすることができた．また，スタックコピーを共有することにより，プログラムによっては高速化されることも分かった．しかし，小さくなったとはいえ，まだオーバヘッドは残っている．ここで用いた処理系はインタプリタ方式であり，処理系自体がそれほど高速であるとはいえない．しかし，最適化されたコンパイラでこの手法を用いると，ライトバリアや生成時バリアによるオーバヘッドが相対的に大きくなる可能性はある．

4.5 関連研究

この節では関連する研究を紹介し議論する．

4.5.1 非局所的脱出

call/cc を非局所的脱出に特化させる研究には call/ep がある [30]．また，SCM などの処理系では call/cc が call/ep のように振る舞うように処理系を作るコンパイルオプションが用意されている．call/ep は，必要になる関数フレームがスタック上に残っている時にのみ呼び出すことのできる継続を生成する．したがってスタックのコピーは発生せず，非局所的脱出の用途で効率良く働く．しかし，call/ep で生成した継続は必要になる関数フレームがスタック上に残っていなければ呼び出すことができないので，call/cc で作った継続より能力が低い．提案手法で call/cc が生成する継続は，呼び出された時に必要になる関数フレームがスタック上からなくなる前にコピーを作るので，従来の継続の能力を保存している．

4.5.2 スタックのコピーの遅延

遅延割り当て（lazy allocation）[31] は高級言語のスタックにオブジェクトを割り当てる実装手法である．スタック上に割り当てたオブジェクトを指すヒープからの参照が作られると，スタック上のオブジェクトをヒープにコピーする．さらに，スタック上のオブジェクトもヒープにコピーしたオブジェクトも同じ方法で操作できるようにするため，スタック上にフォワーディングポインタ（forwarding pointer）を残す．この手法を使えば，関数フレームを，生成時にはスタック上に

割り当て，継続がキャプチャされることにより，ヒープから参照されるようになるとヒープにコピーすることにより，遅延スタックコピー法を同様の効果を得ることができる．しかし，フォワーディングポインタを使うとリードバリア（read barrier）が必要になり，専用ハードウェアなしにはオーバヘッドが大きくなると考えられる．また，スタック上の関数フレームをヒープにコピーするには，スタック上の関数フレームの構造が，処理系実装時に分かっている必要がある．そのため，C言語のスタックをScheme言語のスタックとして利用するような処理系では，C言語のポータビリティを失うことなくこの方式を採用することはできない．実際，文献[31]でも，高性能なScheme処理系での一級継続の問題をすべて解決できるわけではないと指摘している．

一方，提案手法では昇格のタイミングが異なっており，遅延割り当てが大域変数やヒープ上のオブジェクトに参照が作られた時に昇格するのに対し，提案手法では昇格予約フラグをセットするだけで，昇格は継続を生成した関数からリターンするまで遅らされる．これにより，スタックのコピーを複数の継続オブジェクトから共有することが可能になる．また，提案手法ではフォワーディングポインタを必要としないため，実用的な手法であると言える．

4.5.3 SOAR

SOAR[16]は，多くのハードウェアサポート機能を備えたSmalltalk-80[4]言語向けRISCマイクロプロセッサである．Smalltalk-80言語でもScheme言語同様，継続を一級オブジェクトとして扱うことができる．Smalltalk-80言語では，これをコンテキスト（context）と読んでいる．SOAR上でのSmalltalk-80言語の実装は，遅延スタックコピー法と類似しており，継続をキャプチャした関数からリターンするまで，スタックコピーを遅延する．さらに，各関数フレームは，1ビットの参照カウンタ[32]を持っており，遅延スタックコピー法の昇格予約フラグと同様の働きをする．SOARでは，継続オブジェクトがヒープに格納されようとすると，例外が発生し，これにより参照カウンタをセットするタイミングを得ることができる．また，関数からのリターン時にも，その関数フレームの参照カウンタがセットされていれば例外が発生する．このようにSOARでは，遅延スタックコピー法でエスケープバリアやリターンバリアを使って実現している機構がハードウェアの例外により実現されている．それに対して，遅延スタックコピー法は，特殊なハードウェアがなくても実装することができる．

4.6 まとめ

この章では，関数フレームをスタック上に割り当てる処理系における一級継続の実装手法として，遅延スタックコピー法を提案した．遅延スタックコピー法では，継続がキャプチャされても直ちにスタック上の関数フレームのコピーは生成せずに，スタック上の関数フレームへのポインタを持つ継続オブジェクトを生成する．その上で，もし継続をキャプチャした関数からリターンした後も継続が呼び出される可能性が残っていれば，継続をキャプチャした関数からのリターン時に関数フレームをコピーする．ここで，継続が呼び出される可能性が残っているかどうかは，その継続オブジェクトへの参照が，ヒープや大域変数などその関数からリターンした後も寿命が続く可能性のある領域に書き込まれたことがあるかどうかで近似した．また，遅延スタックコピー法で必要となるリターンバリアの実装方法や，スタック戦略により一級継続を実装している処理系でスタックのコピーを共有することにより，効率のよい一級継続を実装する改良も示した．

さらに，これらの手法を実際の処理系に実装し，ベンチマークプログラムを使って性能を評価した．この結果から，本手法は一級継続を使うプログラムを効率良く実行することが分かった．特に，実装の容易さや適用範囲の広さに長所がある一方，効率はあまり良くないスタック戦略と組み合わせることで，スタック戦略の長所を生かしたまま，効率の良い一級継続を実現できることが分かった．以上より，遅延スタックコピー法は関数フレームをスタック上に割り当てる処理系に一級継続を実装する手法として有効であると言える．しかし，ライトバリアや生成時バリアによるオーバヘッドにより，一級継続を使わないプログラムにオーバヘッドがかかる．これについては，第6章で議論する．

第5章　スタックコピー共有法

第4章では，関数フレームのコピーを遅延することにより一級継続を使ったプログラムの効率を高める手法として，遅延スタックコピー法を示した．また，スタック戦略を使った処理系に遅延スタックコピー法を実装する際の改良として，スタックのコピーを共有させる手法も示した．これにより，一部のプログラムでの実行効率の改善が見られた．この章では，さらに多くのプログラムでスタックのコピーを共有することにより一級継続を使ったプログラムの効率を高める手法を提案する．ここで提案する手法を**スタックコピー共有法**（stack copy sharing）と呼ぶ．これは，遅延スタックコピー法をスタック戦略の処理系に実装する際の拡張である．

4.2節では，スタックのコピーを共有することにより，スタック戦略で継続オブジェクトの生成にかかる時間を削減し，また，継続オブジェクトが消費する記憶領域も削減した．例えば，図5.1でc_1とc_2が昇格される場合，通常のスタック戦略では，それぞれの継続オブジェクトが独自のスタックのコピーを持つ．それぞれのスタックのコピーは，スタックボトムからaまでの範囲を含んでいる．そこで，4.2節では，このように，片方の継続オブジェクトが必要とするスタックの範囲が，他方の継続オブジェクトが必要とするスタックの範囲に完全に含まれている場合に限り，両方の継続オブジェクトを同時に昇格させることで，スタックのコピーを共有していた．

しかし，c_1が昇格された後でもc_1のためのスタックのコピーを別の継続オブジェクトと共有できれば，さらに効率が良くなると期待できる．継続c_2をキャプチャしたcall/ccからリターンした後，継続c_1をキャプチャしたcall/ccからリターンする前に別の継続c_3がキャプチャされたとする．このc_3がもし昇格されるなら，その時にコピーするスタックの範囲には，スタックボトムからaまでの範囲が含まれている．この範囲は既にc_1とc_2のためにコピーした領域である．そこで，このスタックのコピーをc_3からも利用することができれば，効率はさらによくなる．

スタックコピー共有法では，このように既に昇格された継続オブジェクトの持つスタックのコピーを，後から昇格される継続オブジェクトからも利用する．この章ではスタックコピー共有法の実装方法を三種類示す．その上で，これらの手法の比較を行う．

以降では，継続オブジェクトcが生成された時のスタックトップのアドレスを

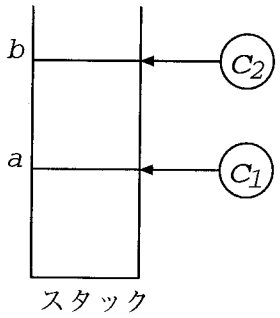

図 5.1: スタックと二つの継続オブジェクト

STADDR(c) と表記することにする.

なお，この章の内容は [24] において発表した内容の一部である.

5.1 実装モデル

この節では，スタックコピー共有法の三種類の実装モデルとして，

- 継木モデル（grafting model）
- スタック分割モデル（dividing stack model）
- インクリメンタル・スタック分割モデル（incremental dividing stack model）

を示す.

5.1.1 継木モデル

継木モデルでは，継続オブジェクトが昇格する際，スタックの過去に一度もコピーされていない範囲だけのコピーを生成し，過去にコピーされている部分については，そのコピーを参照する．また，一個の継続オブジェクトを昇格させる際に，同時にすべての未昇格のアクティブな継続オブジェクトを昇格させ，これらの間でもスタックのコピーを共有する.

図 5.2（a）は，継続 c_1 を含む継続 c_2 の継続オブジェクトを昇格させる様子である．プログラムの実行で最初の昇格では，まだ既に生成されたスタックのコピーがないため，スタック全体をコピーする（斜線部が新たに生成したスタックのコピー）．この時，同時に c_1 も同じスタックのコピーを参照することで昇格させる.

その後，c_1 をキャプチャした `call/cc` からリターンする前にさらに `call/cc` が呼び出され，それによって生成された継続オブジェクト c_3 も昇格されるとする．こ

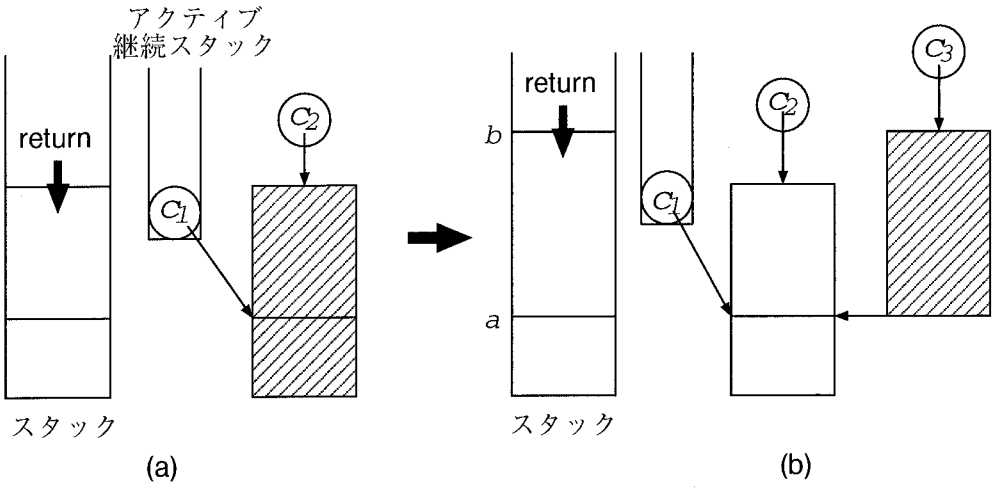

図 5.2: 継木モデル

の時は，図5.2（b）のように，スタックのaからbまでの範囲のみをコピーし，スタックボトムからaまでの範囲は，既にコピーされているものを共有する．

一般の場合では，継続オブジェクトcを昇格させる時，まず，アクティブで，かつ昇格されている継続オブジェクトのうち，最も大きな継続を持つ継続オブジェクトc'を探す．もし条件を満たすc'があれば，スタックのSTADDR(c')とSTADDR(c)の間だけをコピーする．もし，条件を満たすc'がなければ，スタック全体をコピーする．どちらの場合でも，さらに，まだ昇格されていないアクティブな継続オブジェクトを，生成したスタックのコピーを共有することで昇格させる．

ところで，スタック戦略では，継続が呼び出されると，スタックのコピー全体がスタックに書き戻される．スタックのコピーを効率的に共有するためには，この時，アクティブ継続スタックも復元する必要がある．これは，アクティブで，かつ昇格されている継続オブジェクトのうち，最も大きな継続を持つ継続オブジェクトc'をアクティブ継続スタックを使って探すためである．もしアクティブ継続スタックが復元されなければ，継続が呼び出された後ではc'を見つけることができず，スタック全体をコピーすることになってしまう．そこで，継続のキャプチャではスタックのコピーと同様にアクティブ継続スタックも保存する．ただし，我々の実装では，アクティブ継続スタックは継続オブジェクトのリストとして表現してあるため，アクティブ継続スタックの明示的なコピーは発生しない．アクティブ継続スタックの保存が必要な点は，継木モデルだけでなく以降で述べる二つのモデルでも同様である．

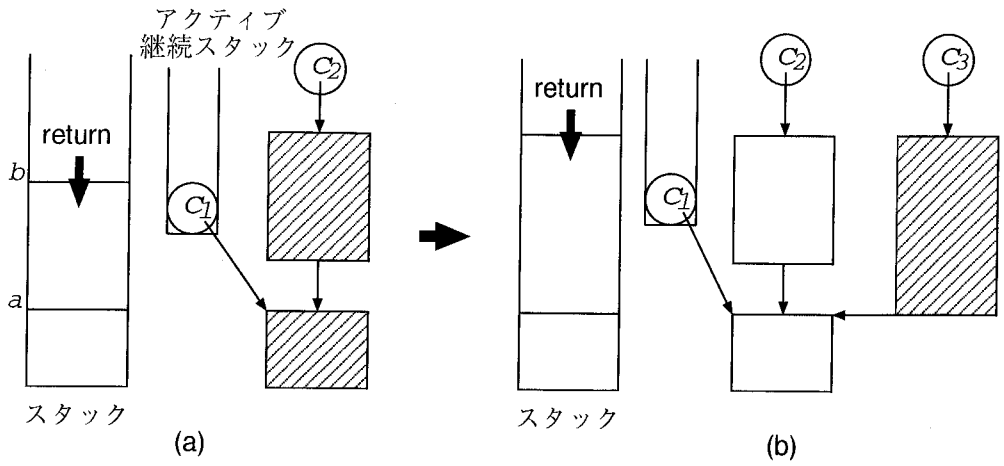

図 5.3: スタック分割モデル

5.1.2 スタック分割モデル

継木モデルでは継続オブジェクトがスタックのコピーの途中のアドレスを持つ可能性がある．例えば，図5.2（a）では継続オブジェクト c_1 はスタックのコピーの途中を指している．このようなオブジェクトの途中のアドレスを指すポインタは，ごみ集めと相性が悪いことがある．スタック分割モデルではスタックを複数の**セグメント**（segment）に分割してコピーする．これによりこのような状況を避けることができる．図5.3は図5.2と同じシナリオで継続オブジェクトがどのように昇格されるかを示している．継続オブジェクト c_2 が昇格される時，スタックは a から b までの範囲と，それより下の範囲の二つのセグメントに分割されてコピーされる（図5.3（a））．スタックの a から b の範囲のコピーは，ボトムから a までの範囲のコピーへのポインタを持っている．これにより，継続 c_2 は，呼び出された時に，スタック全体の内容を正しく復元することができる．その後，継続オブジェクト c_3 が昇格されると，新たなスタックのコピーが生成される（図5.3（b））．しかし，新しく生成されるスタックのコピーは，既にコピーされているボトムから a までの範囲を持たず，ボトムから a までの範囲のコピーへのポインタを持つ．

一般の，n 個のアクティブな継続オブジェクトがある場合を考える．これらを，$c_1, c_2, \ldots, c_n$ とし，$i \leq j$ ならば $c_i \sqsubseteq c_j$ とする．これらは，順にアクティブ継続スタックに格納されている．さらに，ある $i(1 \leq i \leq n)$ について継続オブジェクト $c_j(1 \leq j \leq i-1)$ は昇格されており，$c_k(i \leq k \leq n)$ は昇格されていないはずである．スタック分割モデルでは，c_n を昇格しようとする時，継続スタックを走査して，c_i をみつける．そして，$c_k(i \leq k \leq n)$ について，スタックの $\mathrm{STADDR}(c_{k-1})$ から $\mathrm{STADDR}(c_k)$ までの範囲のコピーを生成し，c_k からそのスタックのコピーを参照するようにする．ただし，$\mathrm{STADDR}(c_0)$ はスタックボトムのアドレスとする．

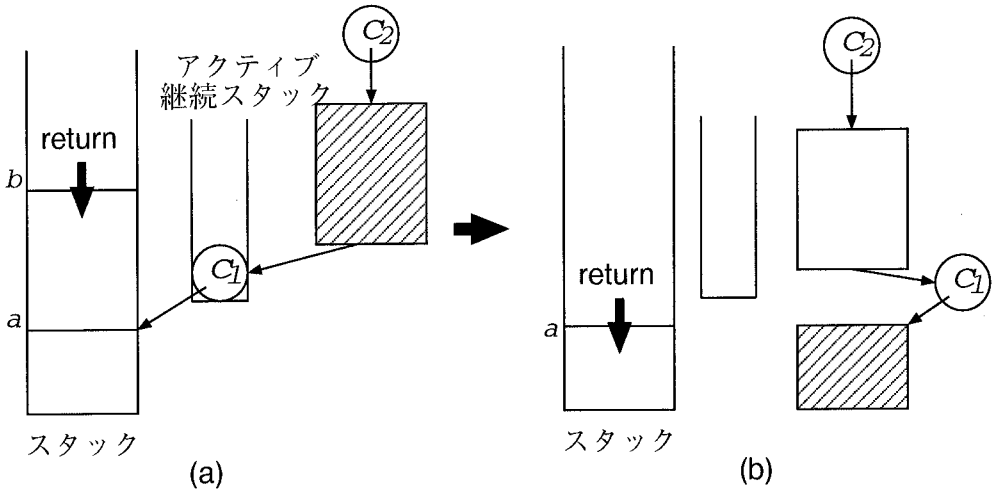

図 5.4: インクリメンタル・スタック分割モデル

また，STADDR(c_{k-1}) から STADDR(c_k) までの範囲のコピー ($2 \leq k \leq n$) には，下の，STADDR(c_{k-2}) から STADDR(c_{k-1}) の範囲のコピーへのポインタを持たせておく．

5.1.3 インクリメンタル・スタック分割モデル

スタック分割モデルでは，ある継続オブジェクトが昇格すると，アクティブな継続オブジェクトのうち未昇格なものをすべて昇格させていた．しかし，実際にはリターンしようとしている `call/cc` で生成された継続オブジェクト以外は，さらに先まで昇格を遅らせても問題は起こらない．インクリメンタル・スタック分割モデルで継続オブジェクトが昇格される様子を図 5.4 に示す．継続オブジェクト c_2 を昇格させる際，スタック分割モデルでは，スタックのボトムから a までの範囲を同時にコピーしていたが，インクリメンタル・スタック分割モデルでは，スタックの a から b までの範囲だけをコピーする（図 5.4 (a)）．また，スタックの a から b までの範囲のコピーは，スタックボトムから a までの範囲のコピーへのポインタを持たない代わりに，継続オブジェクト c_1 へのポインタを持つ．さらに，c_1 には昇格予約フラグをセットし，将来 c_1 を生成した `call/cc` からリターンする際に，スタックボトムから a までの範囲がコピーされるようにしておく．

一般の継続オブジェクト c を昇格させる場合を考える．c を除いて最大のアクティブな継続オブジェクトを c' とする．このような c' が存在すれば，それはアクティブ継続スタックのトップにあるはずである．もし c' が存在すれば，スタックの STADDR(c) から STADDR(c') までの範囲をコピーし，c に生成したスタックコピーを指すポインタを作る．さらに，生成したスタックのコピーから継続オブジェ

クト c' へのポインタを作る．その上で，c' に昇格予約フラグをセットする．これにより，c' や，c' に含まれるアクティブな継続オブジェクトが昇格する際に，残りのスタックがコピーされるようになる．また，もし c 以外にアクティブな継続オブジェクトがなければ，スタック全体をコピーする．c 以外にアクティブな継続オブジェクトがないことは，アクティブ継続スタックが空であるかどうかで判断できる．

5.2 実装モデルの比較

この節ではスタックコピー共有法の三種類の実装モデルを比較する．この節では定性的な比較を行い，ベンチマークプログラムによる実行時間の測定は5.3節で行う．

5.2.1 キャッシュ無効化によるオーバヘッド

一般にスタックのサイズは非常に大きい．スタック分割モデルやインクリメンタル・スタック分割モデルのセグメントのサイズは，スタック上の call/cc を呼び出した関数の関数フレーム間の距離になるが，これも，一般には非常に大きくなる．したがって，スタックのコピーによりそれまでメモリキャッシュ上にあったデータが追い出されてしまう可能性がある．さらに，SPARCプロセッサなどでは，スタックの内容の一部がレジスタにキャッシュされている．このようなアーキテクチャでは，スタックのコピーの際に，レジスタにキャッシュされたスタックの内容をスタックに書き戻さなければならない．このような，キャッシュの無効化や書き戻しは，スタックのコピーの度に発生する．この点では，インクリメンタル・スタック分割モデルは，スタックのコピーを複数回に分割するため，一回でスタック全体をコピーする他のモデルに比べて不利と考えられる．

5.2.2 記憶領域の占有量

インクリメンタル・スタック分割モデルは，それぞれの call/cc からリターンするまで，その call/cc の関数フレームより下位の関数フレームのコピーを生成しない．これにより，例えば，最初に深い再帰呼出しをした後，呼出しの深い位置で計算が続くようなプログラムで call/cc を使う場合，スタックの下位の，計算に関係のない部分のコピーを計算が終わるまで遅らせられる可能性がある．もし，計算が終わるまでスタックの大部分のコピーが遅らせられれば，計算ではその分だけ広い記憶領域を使うことができる．これは，ごみ集めの回数を減らすという

```
(define k-link #f)
(define (f) (g) (f))
(define (g) (call/cc h))
(define (h Y) (i) (call/cc j) #t)
(define (i) (l k-link) #t)
(define (j Z) (set! k-link Z))
(define (l k-link) (call/cc (lambda (X) X)) #t)
(f)
```

図 5.5: 継木モデルではメモリリークするプログラムの例

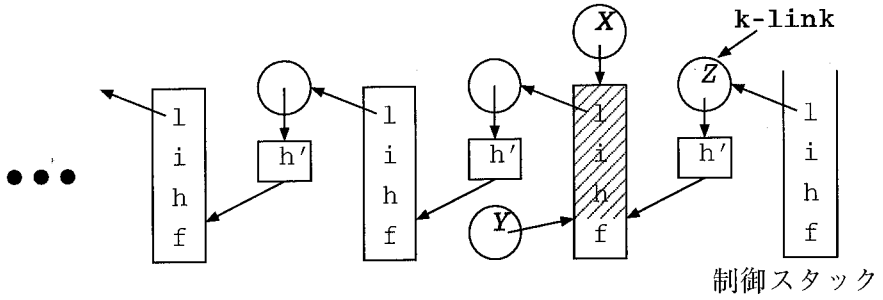

図 5.6: 図5.5により作られる継続オブジェクトの連鎖

点で有利である．しかし，このような特殊なパターンに当てはまるプログラムがそれほど多いとは考えにくく，多くの場合では，ヒープ全体に比べればそれほど大きくないスタックの範囲のコピーが遅らせられるだけに留まると考えられる．

5.2.3 ごみ集めとの相性

継木モデルは，ごみ集めとの相性が悪く，本来回収されるべき継続オブジェクトのメモリが回収されない可能性がある．例えば，図5.5に示すプログラムは，図5.6に示すような，ごみになった継続オブジェクトの連鎖が作られ，ごみになった継続オブジェクトがうまく回収されない．図5.5のプログラムでは，次のようにしてごみになった継続オブジェクトの連鎖が作られる．まず，関数 l が呼び出される．その際，大域変数 k-link に格納されたオブジェクトが引数として渡される．したがって，関数 l の関数フレーム内から k-link に格納されていたオブジェクトへのポインタが作られる．関数 l の中で call/cc が呼び出され，さらに，ここで生成される継続オブジェクト（これを X とする）を昇格させると，継続オブジェクト X の持つスタックのコピーにも大域変数 k-link に格納されていたオブジェクトへの参照が含まれることになる．ところで，継木モデルでは，継続オブジェクト

Xを昇格する時，Xのためにコピーしたスタックのコピーを共有することで，他のアクティブな継続オブジェクトも昇格させる．したがって，たとえ継続オブジェクトXがごみとなっても，この時アクティブな継続オブジェクトがすべてごみにならない限りスタックのコピーがごみとして回収されることはない．実際，継続オブジェクトXはすぐにごみになるが，関数gから呼び出されたcall/ccで生成された継続オブジェクトYはすぐにはごみとはならず，スタックのコピーは回収されない．しかし，関数gで生成した継続は，関数1の実行は含まないため，1に引数として渡されたk-linkの値を使うこともない．このようにしてk-linkに格納されていたオブジェクトへの，決して使われることのない参照が作られる．さらに，この後call/ccが呼び出され，別の継続オブジェクト（これをZとする）が生成される．この継続オブジェクトZは，継続オブジェクトYの持つスタックのコピーの一部を共有する．これにより，継続オブジェクトZもごみにならない限り，継続オブジェクトXを昇格させる時に生成したスタックのコピーも回収されなくなる．この後Zは大域変数k-linkに格納され，次回の関数1の呼び出しで引数として渡されるため，上記の議論が繰り返され，永遠に回収されなくなる．このようにして，本来ごみになるべき継続オブジェクトの連鎖が成長し，回収されないごみが増える．

もしごみ集めプログラムが，スタックのコピーのうち，参照される可能性のある部分だけを選択することができれば，このような問題は起こらない．しかし，このためにごみ集めプログラムを複雑にするのであれば，スタックコピー共有法の別の実装モデルを選択するほうがよいこともある．また，たとえごみ集めプログラムを変更したとしても，最後のスタックのコピーのうち，継続オブジェクトZから使われない部分（図5.6の斜線部分）を回収することはできない．

スタック分割モデルとインクリメンタル・スタック分割モデルでは，スタックのコピーの途中を指すポインタがないため，このような問題は起こらない．これらの実装モデルでは，図5.6の斜線部分は，それ以外の部分と切り離されて，別のオブジェクトとしてコピーされている．そのため，継続オブジェクトXが不要になった時点で，図5.6の斜線部分を回収することができる．

5.3 性能評価

この節では，スタックコピー共有法をScheme言語の処理系に実装し，ベンチマークプログラムを使って性能を測定する．処理系には4.4節でも使用したSCM[14]とMzScheme[15]を用いる．また，ベンチマークプログラムには4.4節で使用したtak，ctak，boyer，puzzle，same-fring（グラフではsfと表示）に加え，ctak/setを用いる．ctak/setの構造はctakと全く同じだが，すべての継続オブジェトが

```
(define (ctak x y z)
  (call/cc (lambda (k) (ctak-aux k x y z))))
(define dummy #f)
(define (ctak-aux k x y z)
  (set! dummy k)
  (cond ((not (< y x))
         (k z))
        (else (call/cc
               (ctak-aux k
                (call/cc (lambda (k) (ctak-aux k (- x 1) y z)))
                (call/cc (lambda (k) (ctak-aux k (- y 1) z x)))
                (call/cc (lambda (k) (ctak-aux k (- z 1) x y)))))))))
```

図 5.7: ctak/set

昇格されるように，ダミーの大域変数への代入を追加してある．ctak/set のプログラムを図 5.7 に示す．

ここでの各処理系への遅延スタックコピー法の実装は，4.4.3 節と同様の洗練された方式になっており，スタックコピー共有法の実装も，洗練された方式による遅延スタックコピー法が実装された処理系をベースにしている[1]．

表 5.1: 生成したスタックコピーの数と合計サイズ

	ctak		puzzle	
	size	object	size	object
original	31922740	47707	7595560	10025
basic	15242992	18171	0	0
grafting	828120	3539	0	0
dividing	828120	13405	0	0
incremental	1029180	13405	0	0
	sf		ctak/set	
	size	object	size	object
original	52800944	400008	31022740	47707
basic	48800864	400008	43022420	47707
grafting	30400496	400006	3300308	34302
dividing	30400496	400008	3300308	47707
incremental	33400556	400008	4015898	47707

[1] MzScheme での実装は皆川宜久氏による．

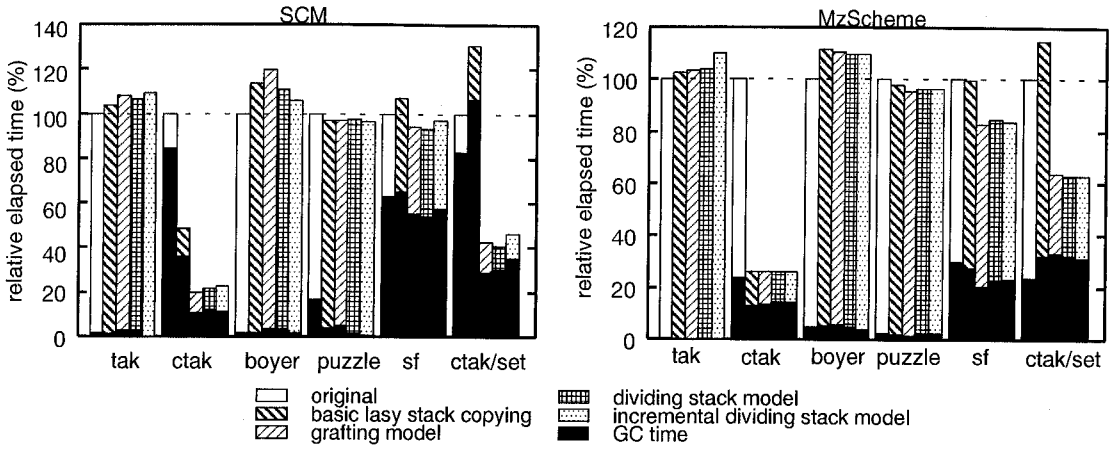

図 5.8: ベンチマークプログラムの相対実行時間

これらの処理系で実行時間を測定した結果を図5.8に示す．図5.8は，スタックコピー共有法を実装していない処理系での実行時間に対する，遅延スタックコピー法のみを実装した処理系（“basic lazy stack copying” と示す）と，三種類の実装モデルによるスタックコピー共有法を実装した処理系での実行時間の比を表している．なお，この測定はPentinum 4を使って行った．また，表5.1に，SCMにおけるプログラムの実行全体でスタックコピーのためにコピーしたスタックのバイト数（“size” で示す）と，生成したスタックコピーのオブジェクト数（“object” で示す）を示す．MzSchemeでも同様の傾向が見られるため，ここではSCMでの結果のみを示す．

結果を見ると，SCM，MzSchemeとも一級継続を使うプログラムでは，スタックコピー共有法も実装した処理系の方が元の処理系よりも良い結果を出し，遅延スタックコピー法のみを実装した処理系と比較しても，同等以上の結果を出している．特に，`same-fring` では，遅延スタックコピー法のみの実装では遅延スタックコピー法を実装していない処理系よりも遅くなっていたが，スタックコピー共有法も実装することにより高速化されている．`same-fringe` は最終的にはすべての継続オブジェクトが昇格され，また，同時に複数の継続オブジェクトが昇格されることもないため，4.2節で述べた方法でもスタックコピーが共有されない．しかし，実際にはスタックのボトム周辺は変化せず，その部分のコピーは複数の継続オブジェクトで共有可能であるため，スタックコピー共有法により共有される．この実験結果はそれを裏付けている．`ctak/set` はこの特徴のみを抽出した人工的なプログラムであるため，同じ傾向が顕著に表れている．このように，スタックコピー共有法によりスタックコピー遅延法のみでは性能が改善できなかったパターンの，一級継続を使ったプログラムも性能を改善することができた．

次にオーバヘッドについて見ると，一級継続を一切使わない `tak` と `boyer` で，元

の処理系と比べて若干のオーバヘッドが見られる．しかし，遅延スタックコピー法と比較すると，処理系やプログラムにより多少の差はあるが，ほぼ同程度のオーバヘッドと見ることができる．スタックコピー共有法を実装した処理系では一級継続を使わない限り，スタックコピー遅延法のみの実装した処理系と比べて，処理系の初期化部分以外で余分なコードは実行されないはずである．それを考えても，オーバヘッドは遅延スタックコピー法とほぼ同等と結論づけてよい．

三種類の実装モデルの性能については，この性能測定で有意な差を得ることはできなかった．表5.1の`ctak`の実行でのスタックコピーのオブジェクト数を見ると，継木モデルよりもインクリメンタル・スタック分割モデルのオブジェクト数4倍程度多くなっている．このことより，スタックコピーの回数が増えることによる，キャッシュ無効化などのオーバヘッドは，今回測定に用いたインタプリタ方式の処理系ではそれほど問題にならないと考えられる．なお，スタック分割モデルでもインクリメンタル・スタック分割モデルと同じオブジェクト数だが，スタック分割モデルでは複数のスタックコピーが同時に生成されるため，スタックコピーの回数は継木モデルと同じ回数である．

5.4 考察

この節では，まず，関数フレームのコピーを複数の継続オブジェクトから共有するという点でスタックコピー共有法と似ている，インクリメンタル・スタック/ヒープ戦略と本手法とを比較する．次に，細粒度のマルチスレッドプログラムに一級継続を利用する場合について考察する．

2.4.3節で紹介したインクリメンタル・スタック/ヒープ戦略[9]は，関数フレームのコピーを複数の継続オブジェクトで共有する一級継続の実装戦略である．一度継続をキャプチャすると，アクティブな関数フレームはヒープ上に関数フレームのリストという形でコピーされる．次回の継続キャプチャの際には，既にコピーしてある関数フレームはリスト操作だけで新しい継続オブジェクトからも共有され，同じ関数フレームを二回以上コピーすることはない．この点は，スタック分割モデルによるスタックコピー共有法とよく似ている．スタック分割モデルでは，関数フレーム単位ではなく，`call/cc`の関数フレームでスタックを区切った，セグメントと呼ぶ単位でスタックをコピーし，やはり，セグメントのリストという形でヒープ上に保持する．両者の違いは，次の二点である．まず，インクリメンタル・スタック/ヒープ戦略では，関数フレーム単位でコピーするため，関数フレームの構造が処理系実装時に分かっていなければならない．それに対して，スタック分割モデルでは，関数フレームやスタックの構造が正確に分からない処理系でも利用できる．これは，1）スタックのセグメントをそのままコピーし，2）実行する時は

最初に生成されたアドレスに書き戻す，また，3）C言語などでスタックポインタの値が正確に分からなくても，局所変数のアドレスで代用しスタックの必要な範囲より少し大きめにコピーすることで可能になる．もう一点は，スタックコピー共有法のスタック分割モデルは遅延スタックコピー法を基にしており，エスケープバリアやリターンバリアのオーバヘッドがある点である．このように，スタックコピー共有法は性能に特化した処理系ではインクリメンタル・スタック/ヒープ戦略に劣る可能性はあるが，幅広い処理系に利用できる点でインクリメンタル・スタック/ ヒープ戦略にはない長所を持っている．

次に，遅延タスク生成（lazy task creation）[33]やその他の細粒度マルチスレッドプログラムの効率を良くする研究[34, 35, 36]がある．コルーチンなどのマルチスレッドプログラムは，一級継続を用いて実現できる．そして，スタックコピー共有法を実装することによりコルーチンのプログラムも従来の処理系より効率が良くなることを示した．しかし，細粒度のマルチスレッドプログラムでは一級継続を使った実装では，オーバヘッドが大きすぎる．そのため，細粒度マルチスレッドが目的のプログラミング言語の処理系では，一級継続だけではなく遅延タスク生成などを使ったマルチスレッド機能も個別に用意することが望ましい．

5.5 まとめ

この章では，スタックコピー共有法を提案した．これは，遅延スタックコピー法を実装したスタック戦略の処理系において，複数の継続オブジェクトの間でスタックのコピーの共有をすることによって，一級継続を使ったプログラムの効率を向上させる手法である．本手法は，インクリメンタル・スタック/ヒープ戦略と異なり，C言語など，スタックやレジスタを直接操作できない言語で記述されたインタプリタや，C言語にコンパイルするコンパイラなどでも利用できるという利点がある．我々は，本手法の三種類の実装モデルを提案し，それぞれの定性的な比較をした．また，ベンチマークプログラムを用いて，本手法を実装していない処理系と実装した処理系で性能を比較し，本手法を実装した処理系が，多くの一級継続を使ったプログラムを効率良く実行することを確認した．さらに，スタックコピー共有法のオーバヘッドは，基となっている遅延スタックコピー法のオーバヘッドのみであることも確認した．

第6章　エスケープバリアの除去

遅延スタックコピー法では，継続オブジェクトがそれを生成したcall/ccより長い寿命を持つ時に継続オブジェクトを昇格させる．第4章の実装では，昇格させるべき継続オブジェクトを効率良く見付けるために，昇格予約フラグを使った．そこでは，継続オブジェクトの参照がそれを生成したcall/ccや，そこから呼び出された関数の関数フレームから外にもれる（これをcall/ccから**エスケープ**（escape）すると呼ぶ）書き込みを見張って，継続オブジェクトに昇格予約フラグをセットする．この仕組みは，生成時バリアとライトバリアにより実現されていた．これらのバリアを合わせて**エスケープバリア**（escape barrier）と呼ぶことにする．第4章の実装では，エスケープバリアは，代入やオブジェクトの生成の処理に埋め込まれていた．そのため，代入やオブジェクト生成であれば常にエスケープバリアが働き書き込まれる値がチェックされていた．第4章で示した性能評価では，このオーバヘッドはそれほど大きくなかった．しかし，遅延スタックコピー法を最適化されたコンパイラに実装する場合，エスケープバリアにより生じるオーバヘッドが相対的に大きくなり，無視できなくなる恐れがある．

しかし，実際にすべてのエスケープバリアで昇格予約フラグをセットしているわけではない．エスケープバリアでは，書き込まれる値をチェックしてそれが昇格されていない継続オブジェクトである時だけ，昇格予約フラグをセットする．これ以外の時はエスケープバリアは実際には有益な処理はしていない．したがって，書き込まれる値が昇格されていない継続オブジェクトでないことが分かっていれば，エスケープバリアを張ってその値をチェックする必要はない．この章では，プログラムを実行前に変形し，必要な箇所に明示的なエスケープバリアが挿入されたプログラムにする手法を提案する．プログラムの変換の際には，データフローを解析して必要がないことが分かったエスケープバリアは除去する．変形されたプログラムは，エスケープバリアを含まないオブジェクト生成や書き込み命令と，チェック命令を持つ処理系によって実行される．これにより不要なエスケープバリアの一部が除去できる．

この章では，まず6.1節で関連する研究を示す．次に6.2節で準備として変換対象となるScheme言語のサブセットを示す．その上で6.3節で核となる変換アルゴリズムを示す．さらに，6.4節で我々のアルゴリズムをフルセットのScheme言語

に適用する方法や，変換によりどの程度のチェックが除去されるかの評価について議論する．また，さらに多くのチェックを除去するための拡張方法についても議論する．

6.1 関連研究

Jong-Deok Choiらは [37] で，Java言語においてオブジェクトがメソッドやスレッドからエスケープしないかどうかを解析する手法を提案している．ある関数の実行中に生成されるオブジェクトがその関数の実行より長い寿命を持たないならば，そのオブジェクトはスタック上に割り付けることができる．一般に，オブジェクトをヒープ上に割り付けるのに比べてスタック上に割り付ける方が効率が良いため，エスケープ解析により一部のオブジェクトをスタックに割り付けて効率良くプログラムを実行することができる．ここでのエスケープ解析で得られる情報は，あるオブジェクトがエスケープする可能性があるかどうかである．一方，我々の目的は，継続オブジェクトの最初のエスケープである可能性のある代入や書き込みを発見することである．そのため，ここでのエスケープ解析をそのまま利用することはできない．

小川らは [38] で，不要なNullポインタチェックを除去する手法を提案している．Java言語などの安全なプログラミング言語では，実行時に処理系がNullポインタチェックを行う必要がある．これは，我々の目的と似ており，同じ内容を持つ変数についての二回目以降のチェックの除去などが目的となっている．[38] でもNullポインタチェックを，従来それが埋め込まれていた命令と分離することで，不要なNullポインタチェックを除去している．[38] では，Nullポインタチェックを実行と逆向きに移動させて冗長なチェックを除去している．この章でも，我々のアルゴリズムに対する拡張として，チェックを実行の逆向きに移動させることにより，冗長なチェックを除去する手法を示す．

ところで，継続オブジェクト以外のオブジェクトに対するチェックを除去する手法として式の型を利用することが考えられる．ML言語などは静的な型を持つため，値が継続オブジェクトとなる可能性がある式かどうかを，型を利用して調べることができる．我々のグループでは，このような手法を用いた不要なエスケープバリア除去の試みも行っている．

6.2 準備

この章では，Scheme言語のサブセットであるScheme0という言語を対象として変換手法を示す．この章で示す手法では，エスケープバリアがプログラム上に明示されていないScheme0言語のプログラムからエスケープバリアを明示したScheme1という言語に変換する．この節ではまず，Scheme0言語を定義し，その後Scheme1言語を定義する．

まず，Scheme0言語の文法を図6.1に示す．ここで$*$は直前にある構文要素の0回以上の繰り返しを表わす．また，⟨variable⟩は変数を表し，⟨literal⟩はリテラルを表す．なお，以降で示す変換アルゴリズムでは，式を$e, e_0, \ldots, e_n$で，変数を$v, v_0, \ldots, v_n$で，リテラルをcで表す．Scheme0言語では，Scheme言語と違い関数呼出し式で引数の数を固定している．また`cons`をキーワードとしている．

Scheme0言語はScheme言語と同じ意味論を持つ．ただし式の評価順序は次のように固定する．関数適用式では，まず関数を評価し，次に引数を評価し，最後に関数を適用する．`letrec`式では，変数の初期化式を出現順に評価する．このようにしても，Scheme言語の仕様には矛盾しない．また，Scheme0言語はトップレベル式を持たないが，`main`の名前を持つ関数が`#t`を引数に呼び出され，プログラムが開始するものとする．つまり，プログラムの最後に

```
(main #t)
```

を追加したScheme言語のプログラムと同じ意味とする．なお，`call/cc`以外にヒープにオブジェクトを生成する組み込み関数を持たないとする．

この言語では，次の箇所でエスケープバリアが必要になる．

- `lambda`式による関数クロージャオブジェクトの生成
- `set!`式による代入
- `cons`式によるセルオブジェクトの生成
- `call/cc`関数

次に，Scheme1言語を図6.2に示す．Scheme1言語の`lambda`式や`set!`式は，引数をチェックしない．また，`cons.nb`式も引数をチェックしない`cons`式である．代わりに，Scheme1言語はチェックのために`check`式を持つ．`check`式は引数をチェックする．`check`式の値は引数に渡された値そのものである．

```
⟨program⟩       ::= ⟨definition⟩*
⟨definition⟩    ::= (define (⟨variable⟩ ⟨variable⟩) ⟨expression⟩)
⟨expression⟩    ::= ⟨variable⟩
                |   ⟨literal⟩
                |   (⟨expression⟩ ⟨expression⟩)
                |   (lambda (⟨variable⟩) ⟨expression⟩)
                |   (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
                |   (letrec (⟨binding spec⟩*) ⟨expression⟩)
                |   (begin ⟨expression⟩*)
                |   (set! ⟨variable⟩ ⟨expression⟩)
                |   (cons ⟨expression⟩ ⟨expression⟩)
⟨binding spec⟩  ::= (⟨variable⟩ ⟨expression⟩)
```

図 6.1: Scheme0 言語

```
⟨program⟩       ::= ⟨definition⟩*
⟨definition⟩    ::= (define (⟨variable⟩ ⟨variable⟩) ⟨expression⟩)
⟨expression⟩    ::= ⟨variable⟩
                |   ⟨literal⟩
                |   (⟨expression⟩ ⟨expression⟩)
                |   (lambda (⟨variable⟩) ⟨expression⟩)
                |   (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩)
                |   (letrec (⟨binding spec⟩*) ⟨expression⟩)
                |   (begin ⟨expression⟩*)
                |   (set! ⟨variable⟩ ⟨expression⟩)
                |   (cons.nb ⟨expression⟩ ⟨expression⟩)
                |   (check ⟨expression⟩)
⟨binding spec⟩  ::= (⟨variable⟩ ⟨expression⟩)
```

図 6.2: Scheme1 言語

6.3 変換アルゴリズム

この節では，次の手順でScheme0言語をScheme1言語へと変換する．まず，consをcons.nbで置き換えると同時に，Scheme0言語のプログラムに暗黙に入っていたエスケープバリアをcheck式により明示する．その後，チェックが必要が値を追跡することにより不要なエスケープバリアを除去する．

6.3.1 エスケープバリアの明示

まず，変換の前半部に相当するconsの置き換えとcheckの挿入の方法を示す．この変換を**T0変換**と呼ぶことにする．checkの挿入が必要になるのは，lambda式，set!式，およびcons.nb式である．このうち，set!式とcons.nb式はそれぞれの引数位置にある式にcheckを挿入すればよい．これに対して，lambda式へのcheckの挿入は自明ではない．以下で詳しく説明する．

lambda式は関数クロージャを生成する．関数クロージャの中では，それが生成された位置で参照できる局所変数を参照することができる．したがって，これらの局所変数のうち，関数クロージャ内で参照される可能性のある変数はチェックしなければならない．関数クロージャ内で参照される変数は，lambda式の自由変数に含まれる．そのため，ここではlambda式の自由変数のうち，lambda式の出現位置で参照できる局所変数に含まれるものをチェックする．lambda式の出現する文脈にかかわらず，局所変数をチェックした後に関数クロージャを生成するために，begin式を使って，次のように局所変数をチェックする．

```
(begin (check ⟨変数⟩) ...
    ⟨部分式を変換したlambda式⟩)
```

ただし，letrec式による再帰的定義の場合は注意が必要である．再帰的定義では，初期化式の計算中はまだ定義されようとしている変数の値は決まっていない．それにもかかわらず，定義されようとしている変数を初期化式中で使うことができる．そのため，前述の方法だけでは，関数クロージャに取り込まれる変数をうまくチェックできない．例として次のプログラムを考える．

```
(define (f k)
  (letrec ((k2  k)
           (clo (lambda (x) k2)))
    clo))
```

このプログラムを前述の方法で変換すると，次のようなプログラムが得られる．

```
(define (f k)
```

```
(letrec ((k2  k)
         (clo (begin (check k2)
                     (lambda (x) k2))))
  clo))
```

このプログラムの関数 f 内では，k2 と clo を定義している．変換後のプログラムでは，clo の初期値の計算で k2 をチェックしているが，まだこの時点では k2 の値は決まっていない．そのため，関数オブジェクト clo から参照される k2 がチェックされないまま関数クロージャオブジェクトが生成される．その結果，f が継続オブジェクトを引数に呼び出されると，その継続オブジェクトがチェックされないまま，それを参照する関数オブジェクト clo が作られてしまう．これに対処するには，初期化式中の lambda 式で同時に定義されようとしている変数のチェックが必要な場合，その初期化式に check を挿入すればよい．前述の例では，次のように変換すれば正しくチェックされる．

```
(define (f k)
  (letrec ((k2  (check k))
           (clo (lambda (x) k2)))
    clo))
```

以上で触れていない式については特に変換の必要はなく，Scheme0 言語のプログラムをそのまま Scheme1 言語のプログラムと読みかえればよい．

以上の変換を図 6.3 にまとめる．ここで，以下の補助関数を使う．

FV(*exp*) *exp* の自由変数の集合を返す．

ChkAll(*varset*) すべての $v \in varset$ について，(check v) を並べた式の列を返す．

ChkIf(*cond*, *exp*) *cond* が真の時は式 (check *exp*) を返し，偽の時は式 *exp* を返す．

図 6.3 の変換では，letrec で定義されようとしている変数の集合を**未定義変数集合** U で保持し，部分式の変換の際に参照できるようにしている．lambda 式に check を挿入する際には，未定義変数集合を参照する．もし未定義変数のチェックが必要でれば，check を挿入する代わりにチェック変数集合 C に含め，その変数を定義しようとしている letrec 式で初期化式に check を挿入できるようにしている．

T0 変換は，Scheme0 言語を直接実行する場合に処理系が自動的に行っていたチェックを明示したに過ぎない．そのため，処理系が自動的にチェックする場合に比べて，余分なチェックやチェック漏れはない．

$$\frac{\mathrm{T0}_{exp}[\![\langle e, \phi\rangle]\!] = \langle e', C\rangle}{\mathrm{T0}_{def}[\![\texttt{(define }(v_0 v_1) e)]\!] = \texttt{(define }(v_0 v_1) e')}$$

$$\frac{}{\mathrm{T0}_{exp}[\![\langle c, U\rangle]\!] = \langle c, \phi\rangle}$$

$$\frac{}{\mathrm{T0}_{exp}[\![\langle v, U\rangle]\!] = \langle v, \phi\rangle}$$

$$\frac{\mathrm{T0}_{exp}[\![\langle e_0, U\rangle]\!] = \langle e'_0, C_0\rangle \quad \mathrm{T0}_{exp}[\![\langle e_1, U\rangle]\!] = \langle e'_n, C_1\rangle}{\mathrm{T0}_{exp}[\![\langle (e_0 e_1), U\rangle]\!] = \langle (e'_0 e'_1), C_0 \cup C_1\rangle}$$

$$\frac{\mathrm{T0}_{exp}[\![\langle e, U\rangle]\!] = \langle e', C\rangle \quad F = \mathrm{FV}\,(\texttt{(lambda }(v) e))}{\begin{array}{l}\mathrm{T0}_{exp}[\![\langle \texttt{(lambda }(v) e), U\rangle]\!] = \\ \qquad \langle \texttt{(begin ChkAll}\,(F \backslash U)\,\texttt{(lambda }(v) e')), F \cap U\rangle\end{array}}$$

$$\frac{\mathrm{T0}_{exp}[\![\langle e_0, U\rangle]\!] = \langle e'_0, C_0\rangle \quad \ldots \quad \mathrm{T0}_{exp}[\![\langle e_2, U\rangle]\!] = \langle e'_2, C_2\rangle}{\mathrm{T0}_{exp}[\![\langle \texttt{(if } e_0 e_1 e_2), U\rangle]\!] = \langle \texttt{(if } e'_0 e'_1 e'_2), \cup_{i=0}^{2} C_i\rangle}$$

$$\frac{\begin{array}{c}\mathrm{T0}_{exp}[\![\langle e_1, U \cup \{v_1 \ldots v_n\}\rangle]\!] = \langle e'_1, C_1\rangle \ldots \\ \mathrm{T0}_{exp}[\![\langle e_n, U \cup \{v_1 \ldots v_n\}\rangle]\!] = \langle e'_n, C_n\rangle \\ C = \cup_{i=1}^{n} C_i \\ \mathrm{T0}_{exp}[\![\langle e_{body}, U \backslash \{v_1 \ldots v_n\}\rangle]\!] = \langle e'_{body}, C_{body}\rangle\end{array}}{\begin{array}{l}\mathrm{T0}_{exp}[\![\langle \texttt{(letrec } ((v_1 e_1) \ldots (v_n e_n)) e_{body})\rangle]\!] = \\ \langle \texttt{(letrec } ((v_1\ \mathrm{ChkIf}\,(v_1 \in C, e'_1)) \ldots (v_n\ \mathrm{ChkIf}\,(v_n \in C, e'_n)))\, e'_{body}), \\ \qquad\qquad C \cup C_{body} \backslash \{v_1 \ldots v_n\}\rangle\end{array}}$$

$$\frac{\mathrm{T0}_{exp}[\![\langle e_1, U\rangle]\!] = \langle e'_1, C_1\rangle \quad \ldots \quad \mathrm{T0}_{exp}[\![\langle e_n, U\rangle]\!] = \langle e'_n, C_n\rangle}{\mathrm{T0}_{exp}[\![\langle \texttt{(begin } e_1 \ldots e_n), U\rangle]\!] = \langle \texttt{(begin } e'_1 \ldots e'_n), \cup_{i=1}^{n} C_i\rangle}$$

$$\frac{\mathrm{T0}_{exp}[\![\langle e, U\rangle]\!] = \langle e', C\rangle}{\mathrm{T0}_{exp}[\![\langle \texttt{(set! } v\ e), U\rangle]\!] = \langle \texttt{(set! } v\ \texttt{(check } e')), C\rangle}$$

$$\frac{\mathrm{T0}_{exp}[\![\langle e_0, U\rangle]\!] = \langle e'_0, C_0\rangle \quad \mathrm{T0}_{exp}[\![\langle e_1, U\rangle]\!] = \langle e'_1, C_1\rangle}{\mathrm{T0}_{exp}[\![\langle \texttt{(cons } e_0 e_1), U\rangle]\!] = \langle \texttt{(cons.nb (check } e'_0)\texttt{(check } e'_1)), C_0 \cup C_1\rangle}$$

図 6.3: T0 変換:チェックの挿入

6.3.2 エスケープバリアの除去

6.3.1節では，Scheme0言語のプログラムにcheck式を挿入することによりScheme1言語に変換した．ここでは，Scheme1言語に変換されたプログラムから不要なチェックを除去する．この変換を**T1変換**と呼ぶことにする．

まずT1変換を図6.4と図6.5に示す．ここでは，プログラム中に現れる変数はそれぞれ別の名前を持つように名前を付けかえているものとしている．また，変換で使われる補助関数ResultVariableについては，図6.6で示す．以下では，Scheme1言語の式を次のように分類し，それぞれの変換方法について説明する．

- 部分式の値が式全体の値となる式（if, begin, letrec）
- リテラルとオブジェクト生成式（lambda, cons.nb）
- 値が定義されていない式（set!）
- 変数参照式
- 関数呼出し式

まず，部分式の値が式全体の値となる式に対して，その値がチェックの対象となっている場合を考える．このような式の値のチェックは，そのままでは不要なチェックかどうかを判断することが難しい．しかし，このようなチェックを式の値となる可能性のある部分式に移しても，チェックされる値は変わらない．例えば次の式ではif式全体の値がチェックの対象となっている．

```
(check (if (not (null? x))
           x
           (cons.nb (check y) (check z))))
```

この式から各部分式をチェックするように書き換えた式を次に示す．

```
(if (not (null? x))
    (check x)
    (check (cons.nb (check y) (check z))))
```

このように変換することで，チェックが本当に必要かどうかを判断できるようになる．

次に，リテラルとオブジェクト生成式を考える．継続オブジェクトはリテラルで表現することはできない．また，lambda式，cons.nb式はそれぞれ関数クロージャオブジェクトとセルオブジェクトを生成するため，チェックは不要である．したがって，これらの式の値をチェックしているcheck式は除去できる．

$$\frac{\mathrm{T1}_{exp}[\![\langle e, \{\{v_1\}\}\rangle]\!] = \langle e', C\rangle}{\mathrm{T1}_{def}[\![\texttt{(define (}v_0 v_1\texttt{)}e\texttt{)}]\!] = \texttt{(define (}v_0 v_1\texttt{)}e'\texttt{)}}$$

$$\frac{}{\mathrm{T1}_{exp}[\![\langle c, C\rangle]\!] = \langle c, C\rangle}$$

$$\frac{}{\mathrm{T1}_{exp}[\![\langle v, C\rangle]\!] = \langle v, C\rangle}$$

$$\frac{\mathrm{T1}_{exp}[\![\langle e_0, C\rangle]\!] = \langle e'_0, C'\rangle \quad \mathrm{T1}_{exp}[\![\langle e_1, C'\rangle]\!] = \langle e'_n, C''\rangle}{\mathrm{T1}_{exp}[\![\langle (e_0 e_1), C\rangle]\!] = \langle (e'_0 e'_1), C''\rangle}$$

$$\frac{\mathrm{T1}_{exp}[\![\langle e, \{\{v\}\}\rangle]\!] = \langle e', C'\rangle}{\mathrm{T1}_{exp}[\![\langle \texttt{(lambda (}v\texttt{)}e\texttt{)}, C\rangle]\!] = \langle \texttt{(lambda (}v\texttt{)}e'\texttt{)}, C\rangle}$$

$$\frac{\begin{array}{c}\mathrm{T1}_{exp}[\![\langle e_0, C\rangle]\!] = \langle e'_0, C'\rangle \\ \mathrm{T1}_{exp}[\![\langle e_1, C'\rangle]\!] = \langle e'_1, C_1\rangle \\ \mathrm{T1}_{exp}[\![\langle e_2, C'\rangle]\!] = \langle e'_2, C_2\rangle\end{array}}{\mathrm{T1}_{exp}[\![\langle \texttt{(if } e_0 e_1 e_2\texttt{)}, C\rangle]\!] = \langle \texttt{(if } e'_0 e'_1 e'_2\texttt{)}, C_1 \cup C_2\rangle}$$

$$\frac{\begin{array}{l} C_0 = \{\{v\} \mid (v, e) \in \{(v_1, e_1) \ldots (v_n, e_n)\}, s \in C, \mathrm{ResultVariable}(e) \cap s \neq \phi\} \\ \cup \{s' \mid s \in C, s' \text{ is a minimum set s.t. } s' \supseteq s, \text{if } e_i \in s \text{ then } v_i \in s' (1 \leq i \leq n)\} \\ \mathrm{T1}_{exp}[\![\langle e_1, C_0\rangle]\!] = \langle e'_1, C_1\rangle \quad \ldots \quad \mathrm{T1}_{exp}[\![\langle e_n, C_{n-1}\rangle]\!] = \langle e'_n, C_n\rangle \\ \mathrm{T1}_{exp}[\![\langle e_{body}, C_n\rangle]\!] = \langle e'_{body}, C_{body}\rangle \end{array}}{\begin{array}{l}\mathrm{T1}_{exp}[\![\langle \texttt{(letrec ((}v_1 e_1\texttt{)} \ldots \texttt{(}v_n e_n\texttt{))}e_{body}\texttt{)}, C\rangle]\!] = \\ \qquad \langle \texttt{(letrec ((}v_1\ e'_1\texttt{)} \ldots \texttt{(}v_n\ e'_n\texttt{))}e'_{body}\texttt{)}, C_{body}\rangle\end{array}}$$

$$\frac{C_0 = C \quad \mathrm{T1}_{exp}[\![\langle e_1, C_0\rangle]\!] = \langle e'_1, C_1\rangle \quad \ldots \quad \mathrm{T1}_{exp}[\![\langle e_n, C_{n-1}\rangle]\!] = \langle e'_n, C_n\rangle}{\mathrm{T1}_{exp}[\![\langle \texttt{(begin } e_1 \ldots e_n\texttt{)}, C\rangle]\!] = \langle \texttt{(begin } e'_1 \ldots e'_n\texttt{)}, C_n\rangle}$$

$$\frac{\mathrm{T1}_{exp}[\![\langle e, C\rangle]\!] = \langle e', C'\rangle}{\mathrm{T1}_{exp}[\![\langle \texttt{(set!}\ v\ e\texttt{)}, C\rangle]\!] = \langle \texttt{(set!}\ v\ e'\texttt{)}, \{s' \mid s \in C', s' = s \backslash \{v\}\}\rangle}$$

$$\frac{\mathrm{T1}_{exp}[\![\langle e_0, C\rangle]\!] = \langle e'_0, C_0\rangle \quad \mathrm{T1}_{exp}[\![\langle e_1, C_0\rangle]\!] = \langle e'_1, C_1\rangle}{\mathrm{T1}_{exp}[\![\langle \texttt{(cons.nb } e_0 e_1\texttt{)}, C\rangle]\!] = \langle \texttt{(cons.nb } e'_0 e'_1\texttt{)}, C_1\rangle}$$

図 6.4: T1 変換:チェックの除去

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ c), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle c, C\rangle]\!]$$

$$\frac{v \in \bigcup C}{\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ v), C, R\rangle]\!] = \langle(\texttt{check}\ v), \{s | s \in C, v \notin s\}\rangle}$$

$$\frac{v \notin \bigcup C}{\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ v), C\rangle]\!] = \langle v, C\rangle}$$

$$\frac{\mathrm{T1}_{exp}[\![\langle(e_0 e_1), C\rangle]\!] = \langle e', C'\rangle \quad \mathrm{ResultVariable}((e_0 e_1)) \cap \bigcup C \neq \phi}{\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (e_0 e_1)), C\rangle]\!] = \langle(\texttt{check}\ e'), C'\rangle}$$

$$\frac{\mathrm{T1}_{exp}[\![\langle(e_0 e_1), C\rangle]\!] = \langle e', C'\rangle \quad \mathrm{ResultVariable}((e_0 e_1)) \cap \bigcup C = \phi}{\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (e_0 e_1)), C\rangle]\!] = \langle e', C'\rangle}$$

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (\texttt{lambda}\ (v) e)), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle(\texttt{lambda}\ (v) e), C\rangle]\!]$$

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (\texttt{if}\ e_0 e_1 e_2)), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle(\texttt{if}\ e_0 (\texttt{check}\ e_1)\ (\texttt{check}\ e_2)), C\rangle]\!]$$

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (\texttt{letrec}\ ((v_1 e_1) \ldots (v_n e_n)) e_{body})), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle(\texttt{letrec}\ ((v_1 e_1) \ldots (v_n e_n))\ (\texttt{check}\ e_{body})), C\rangle]\!]$$

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (\texttt{begin}\ e_1 \ldots e_n)), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle(\texttt{begin}\ e_1 \ldots e_{n-1}\ (\texttt{check}\ e_n)), C\rangle]\!]$$

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (\texttt{set!}\ v\ e)), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle(\texttt{set!}\ v\ e), C\rangle]\!]$$

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (\texttt{cons.nb}\ e_0 e_1)), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle(\texttt{cons.nb}\ e_0 e_1), C\rangle]\!]$$

$$\mathrm{T1}_{exp}[\![\langle(\texttt{check}\ (\texttt{check}\ e)), C\rangle]\!] = \mathrm{T1}_{exp}[\![\langle(\texttt{check}\ e), C\rangle]\!]$$

図 6.5: T1 変換:チェックの除去（続き）

次に，値が定義されていない式について考える．プログラムはこのような式の値に依存しないはずなので，たとえ継続オブジェクトが値になるような処理系の実装であったとしても，この継続オブジェクトが呼び出されることはないはずである．したがって，この式の値のチェックも不要であり，除去できる．

最後に，変数参照式と関数呼出し式について考える．ここでは，まだチェックされていない未昇格の継続オブジェクトを，**安全でない値**と呼ぶことにする．安全でない値は一度チェックされると安全な値になる．関数の実行の最初で参照できる変数には，大域変数，クロージャに閉じ込められた変数，および引数がある．このうち，大域変数とクロージャに閉じ込められた変数の値は既にチェックされている．そのため，関数の実行の最初で安全でない値を持つ可能性のある変数は引数のみである．そこで，引数の値を追跡し，この値の最初のチェックとなる可能性のある `check` 式のみを残し，それ以外の `check` 式を除去する．T1 変換では，関数の各点で**安全でない値の集合** C を考え，安全でない値を追跡する．安全でない値の集合の要素は，その値を持つ変数の集合である．したがって，関数の実行の最初の時点での安全でない値の集合は，それぞれの引数について生成されたシングルトンから成っている．

次に，関数の実行が進むことにより，安全でない値の集合がどのように変化するかを考える．ここでは，関数呼出し式の返り値が安全かどうかを判断する必要がある．これには，「関数呼出しの引数がすべて安全であれば，返り値は安全である」という事実を利用する．これは次の理由による．関数呼出しの返り値は，関数呼出しにより渡された引数か，または呼び出された関数内で生成された値である．このうち，関数呼出しにより渡された引数は仮定より安全である．また，呼び出された関数内で継続オブジェクトが生成されたとしても，もしその継続オブジェクトが返り値となるなら，関数からリターンする際に昇格される．したがって，関数内で生成された値が返り値となる場合も返り値は安全である．この事実を変換で利用するために，それぞれの式についてその式が取りうる値を持つ変数の集合を考える．これを**結果変数集合**と呼ぶ．結果変数集合は，直観的には，式の値となる部分式に現れる変数の集合である．正確には図 6.6 に示す方法で求まる．関数呼出し式では，結果変数集合の中のどれかの変数が安全でない値を持つ時，関数呼出し式の結果が安全でない可能性がある．

Scheme1 言語では，`set!` 式による代入と，`letrec` 式による新しい変数の導入で，引数の値が別の変数に格納される可能性がある．しかし，`set!` 式では代入時に代入される値がチェックされるため，それ以降の追跡は必要ない．したがって，`letrec` 式により導入される変数のみについて考えれば良い．`letrec` 式により導入される変数が安全でない値を持つ可能性があるのは，その初期化式が

$$
\begin{aligned}
&\mathrm{ResultVariable}(c) = \phi \\
&\mathrm{ResultVariable}(v) = \{v\} \\
&\mathrm{ResultVariable}((e_0 e_1)) = \mathrm{ResultVariable}(e_1) \\
&\mathrm{ResultVariable}((\texttt{lambda}\ (v) e)) = \phi \\
&\mathrm{ResultVariable}((\texttt{if}\ e_0 e_1 e_2)) = \mathrm{ResultVariable}(e_1) \cup \mathrm{ResultVariable}(e_2) \\
&\mathrm{ResultVariable}((\texttt{letrec}\ ((v_1 e_1) \ldots (v_n e_n)) e_{body})) = \mathrm{ResultVariable}(e_{body}) \\
&\mathrm{ResultVariable}((\texttt{begin}\ e_1 \ldots e_n)) = \mathrm{ResultVariable}(e_n) \\
&\mathrm{ResultVariable}((\texttt{set!}\ \ v\ e)) = \phi \\
&\mathrm{ResultVariable}((\texttt{cons.nb}\ e_0 e_1)) = \phi \\
&\mathrm{ResultVariable}((\texttt{check}\ e)) = \mathrm{ResultVariable}(e)
\end{aligned}
$$

図 6.6: 結果変数集合

1. 安全でない値を持つ可能性のある変数の変数参照の場合，または
2. 関数呼出し式で，その返り値が安全でない値の可能性がある場合

である．1の場合，導入された変数は必ず初期化式の変数と同じ値を持つ．したがって，導入された変数と初期化式の変数のどちらか一方でもチェックされると，他方のチェックも不要になる．そこで，導入された変数を初期化式に現れる変数と同じ値を持つ集合に加える．一方，2の場合は状況が異なる．前述の事実より，関数の返り値が安全でないのは，引数に与えられたどれかの安全でない値が返される場合である．しかし，返り値が必ず引数の値であるわけではないため，返り値をチェックしたからといって引数に渡した値のチェックが不要になるわけではない．そこで，2の場合では新しく安全でない値が生成されたと考える．変換の操作としては，新しく導入された変数からなるシングルトンを，安全でない値の集合に追加する．

`check`式により一度チェックされた値は，それ以降チェックする必要がなくなるため，安全でない値の集合から取り除くことができる．また，`set!`式により別の値が代入された変数は，代入前の値を持たなくなるため，その値を持つ変数の集合から取り除くことができる．

6.4 議論

この節では，まず6.3節で示した変換アルゴリズムを，フルセットのScheme言語が変換できるように拡張する．次に，拡張されたアルゴリズムを使って実際に

いくつかのプログラムを変換し，どの程度エスケープバリアが除去できたかを調べる．最後に，6.3 節で示したアルゴリズムを拡張し，より多くのエスケープバリアが除去できるようにする手法を二つ挙げる．

6.4.1 Scheme 言語の変換

Scheme0 言語は Scheme 言語のサブセットであり，Scheme 言語に次の制限を加えた．

- 構文の種類の制限
- 引数の数の固定
- 関数の引数や `letrec` 式の初期化式の評価順序の固定
- オブジェクトの種類の制限
- オブジェクトのスロットへの代入の制限
- `cons` をキーワードとする制限

実際の Scheme 言語で利用するにはこれらの制限を取り除くか，Scheme 言語をこれらの制限されたプログラムに変換すればよい．

まず構文の種類については，変数名の付け換えや Scheme0 言語の構文の組み合わせによって，Scheme0 言語にない Scheme 言語の構文を表現することができる．ただし，ループ構文については Scheme0 言語の構文で表現すると関数クロージャが生成されてしまうため，アルゴリズムの修正で対処する方がよい．T0 変換では，ループの制御変数の再初期化式を代入と考えてチェックを挿入する．T1 変換では，制御変数の初期化式を新しい変数の導入と考えて，`letrec` 式と同様の処理をする．また，ループ本体の先頭での安全でない値の集合は，制御変数の導入により拡張された集合とすればよい．

引数の数は変換アルゴリズムを修正することで容易に任意の数の引数に対応することができる．引数の数については，Scheme 言語の関数をカリー（Curry）化した上で変換するという方法も可能だが，カリー化により関数クロージャが生成されるため，変換アルゴリズムを修正することで対応する方が良い．

評価順序については，ほとんどの処理系で処理系ごとに評価順序が決まっているため，問題とはならない．もし評価順序が決まっていない場合でも，一時変数を導入することで評価順序を強制することができる．

オブジェクトの種類については，Scheme言語のそれぞれのオブジェクトを生成する関数についてcons.nbと同様にエスケープバリアを持たないバージョンを用意すればよい．

オブジェクトのスロットへの代入についてもエスケープバリアを持たないバージョンを用意し，代入される値を明示的にチェックするようにすればよい．

consがキーワードとなっている点は，consを引数として関数に渡したり，変数に代入する場合に問題となる．そこで，Scheme1言語にconsとcons.nbの二つの関数を追加し，cons.nb式を特殊形式ではない，通常の関数呼出し式とする．その上で，Scheme0言語の関数呼出し式の関数位置にあるconsのみをcons.nbに置き換えることにする．これにより，直接consと記述された関数呼出しにかかっているエスケープバリアは除去の対象となる．一方，関数位置以外のconsはエスケープバリアを含んだconsのまま残るため，これらのconsが呼び出されることがあってもチェックが漏れることはない．

6.4.2 性能評価

ここでは，いくつかのScheme言語のプログラムを実際に変換し，その効果を調べた結果を示す．変換したプログラムは第4章でも性能評価に用いたboyer，puzzle，same-fringeである．takとctakにはエスケープバリアが一切ないため，ここでの実験には用いなかった．表6.1は実験の結果をまとめた表である．表のconsはconsによるセルオブジェクトの生成回数を表しており，set，setobjはそれぞれ変数への代入とオブジェクトのスロットへの代入の回数を表している．なお，puzzleではセルオブジェクト以外にベクタオブジェクトも生成する．しかし，ベクタオブジェクトの生成はプログラム読み込み時に一度だけ行い，性能評価を行う際には測定の対象外とされることが多い．そのため，この実験でも測定の対象外とした．これらの値を使って，次の式によりエスケープバリアが明示されていないプログラムを直接実行する処理系が，実行時にエスケープバリアによるチェックを行う回数が計算できる．この計算結果を表のruntimeの列に示す．

$$\langle \text{runtime} \rangle = \langle \text{cons} \rangle \times 2 + \langle \text{set} \rangle + \langle \text{setobj} \rangle$$

表6.1: エスケープバリアの除去によるチェック回数の変化

プログラム	mkobj2	set	setobj	runtime(a)	check(b)	b/a
boyer	226,464	126,345	0	579,273	437,996	0.77
puzzle	0	4,100	34,857	38,957	6,022	0.15
same-fringe	2	40,006	0	40,010	40,008	1.00

また，checkはこの章で述べた手法により変換したプログラムでcheck式が実行された回数である．さらに，本手法によるチェックの除去の効果であるruntimeとcheckの比をb/aの列に示す．

以下で実験の結果を考察する．boyerでは，継続オブジェクトを一切生成していないため，エスケープバリアが実際には一切必要ない．しかし，実際に除去できたエスケープバリアは23％程度しかない．この原因は，1）ほとんどのチェック対象が関数の返り値となっていたためと，2）プログラムが小さな関数に分割されていたためと考えられる．関数の返り値がチェックの対象となる場合でも，関数の引数をそのまま返す関数は実際には少なく，boyerでもそのような例はなかった．しかし，一般的には引数がそのまま返される可能性を排除できず，エスケープバリアを残すしかない．さらに，返り値をチェックしても安全でない値の集合は小さくならず，実際には同じ値が重複してチェックされる可能性がある．また，6.3節のアルゴリズムでは関数の最初にすべての引数を安全でない変数と仮定しているため，関数が小さいと効果は小さい．このような弱点に対する改良は6.4.4節で述べる．

次にpuzzleについては，ほとんどのエスケープバリアが除去されている．この理由としては，第一にpuzzleが大域変数を多用しており，またリテラルを多く含むため，うまく不要なエスケープバリアが除去されたためと考えられる．第二に，puzzleで多く使われている代入により安全でない値集合が小さくなり，二重のチェックが除去されたためと考えられる．

最後にsame-fringeを見ると，ほとんどエスケープバリアが除去されていない．しかし，same-fringeのプログラムでは，実際に40,008個の継続オブジェクトが生成されており，それらは昇格が必要であった．つまり，same-fringeでは不要なエスケープバリアはなかったため，除去されなかった．

以上より，6.3節のアルゴリズムは関数が小さいプログラムや関数の返り値がチェックの対象となるようなプログラムでは，そのままでは効果が薄いが，それ以外のプログラムにはある程度の効果をあげていると言える．

6.4.3 エスケープバリアの移動による除去

[38]では，Nullポインタチェックを実行と逆方向に移動させることにより，冗長なチェックを除去している．エスケープバリアの除去でも，同様に実行と逆方向に移動させることにより冗長なチェックを除去できる．

6.3節で述べたアルゴリズムでは，関数呼出し式の返り値をチェックしても，その引数のオブジェクトを安全な値にはできない．次のプログラムを考える．

```
(define (f x)
```

```
(set! y (g x))
(set! z x))
```

このプログラムでは，二回大域変数への代入が行われている．これを6.3節のアルゴリズムで変換すると，次のプログラムが得られる．

```
(define (f x)
  (set! y (check (g x)))
  (set! z (check x)))
```

しかし，もしxのチェックを関数gの呼出しよりも前に移動させれば，チェックは一回でよい．このように，変数のチェックを実行の流れに沿って実行と逆向きに移動させることができれば，関数呼出し式の返り値のチェックを除去できる可能性がある．

チェックを移動させる変換を6.3節で述べたアルゴリズムに追加するには次のようにすればよい．まず，T1変換をリテラル式，オブジェクト生成式，および部分式の値が式の値となる式に関する変換と，それ以外の式に関する変換の二つに分ける．その上で，前者の変換をまず行い，次に変数のチェックを移動させる．最後に後者の変換を行う．

変数のチェックの移動では，代入を行うset!式に注意する必要がある．変数vのチェックは，vへの代入式や，それを部分式に持たない式を飛び越えて，実行と逆方向に移動させることができる．しかし，vへの代入式を部分式に持つ式を飛び越えて移動させることはできない．これは，もし代入式やそれを部分式に持つ式を飛び越えてvのチェックを移動させてしまった場合，チェックの対象が変わってしまう可能性があるためである．そこで，vのチェックを，vへの代入を部分式に持つ式の直後まで移動させた後は，その部分式の最後にvのチェックを移動させる．その上で，改めて実行の逆方向に移動させる．また，vのチェックがvへの代入式の直後まで移動した時は，そのvのチェックは直ちに除去してよい．なぜなら，代入されるオブジェクトは代入式によりチェックの対象になる（つまり，check式が挿入される）からである．

次に，条件分岐についても考慮する必要がある．if式による条件分岐では，実行の逆方向に移動させると，二つの部分式から変数のチェックが合流することになる．ここで，両方の分岐でチェックの対象となっている変数のみが，合流することができる．それ以外の変数は分岐直後にチェックされるように，その位置にチェックを留めなければならない．もし，片方の分岐でしかチェックされない変数も分岐前にチェックしてしまうと，冗長なチェックが行われる可能性がある．これにより，このようなチェックの移動を行わなければ昇格されないはずの継続オブジェクトが昇格させる可能性がある．これは，遅延スタックコピー法の効果を打ち消すため，望ましくない．

以上の点に注意すれば，変数のチェックを実行の流れに沿って，実行と逆方向に移動させることができる．これにより，関数呼出し式の返り値のチェックを一部除去することが可能になる．

6.4.4 関数間の情報利用によるエスケープバリアの除去

6.4.2節で挙げた弱点は，次のようにして克服することができる．まず，6.3節のアルゴリズムを変形して，組み込み関数に関する知識を使うようにする．ほとんどの組み込み関数は引数に渡された値をそのまま返すことはない．そこで，これらの関数の返り値を安全な値とすることで，多くのエスケープバリアが除去できる．実際にboyerについてこれを行うと，チェックの回数は65,210回となり，処理系が実行時にチェックする場合に比べ11％しかチェックが行われなくなった．

さらに，ユーザ定義の関数についても同様の手法を考えることができる．ユーザ定義関数の本体の式全体の結果変数集合に含まれる変数が，関数の最後での安全でない値を持っていなければ，関数は安全でない変数を返さない．例えば次の関数の本体の式全体の結果変数集合にはyとzが含まれるが，関数の最後で安全でない値を持つ変数はxだけである．したがって，この関数はどのような引数を受けとっても，安全な値しか返さない．

```
(define (f x)
  (letrec ((y 1))
    (if x y z)))
```

ただし，ユーザ定義関数の返り値を分類する際には次の点に注意しなければならない．まず再帰呼出しを行っている関数では，関数の返り値が安全かどうか決定できる前にその結果を使わなければならない．これには，プログラム中のすべての関数n個について，返り値が安全であるか安全でないかの状態を持つ2^n個の要素を持つ集合上の最小不動点を求めることで対処できる．次に，関数が再定義される可能性を考慮しなければならない．これには，保守的な立場をとって対処する．つまり，再定義されている関数は安全でない値を返すとする．しかし，プログラムを分割してコンパイルする場合には，関数が再定義されているかどうか分からないため，プログラマが再定義されない関数をプログラムに明示するような仕組を作るなどの必要がある．

プログラムが小さな関数に分割されている場合に十分に効果が発揮されないという弱点も，ユーザ定義関数の返り値を分類することである程度克服できる．しかし，返り値ばかりではなく引数についても関数間で安全かどうかの情報を受け渡すことで，さらに多くの不要なエスケープバリアを除去することができる．6.3節のアルゴリズムでは，関数の先頭ではすべての引数が安全でないと仮定してい

た．しかし，実際には多くの引数が安全と分かっていることの方が多い．そこで，一つのn引数の関数定義をテンプレートとして使い，2^n個の関数定義を生成する．それぞれの関数定義はテンプレートの関数定義を，関数の先頭でそれぞれの引数が安全かどうかについて異なる仮定をおいて変換した関数定義である．その上で，関数呼出し式では関数に渡す引数が安全な値を持つかどうかにより適切な関数を呼び出すようにする．ただし，関数を一級オブジェクトとして変数に代入したり，高階な関数に引数として渡す時は，すべての引数が安全でないと仮定したバージョンの関数を渡すようにする．これにより，`call/cc`にも引数が安全でないと仮定したバージョンが渡されることになる．このようにすると，高階な関数を使っていない`boyer`ではすべてのエスケープバリアが除去できた．また，`puzzle`も`call/cc`以外に高階な関数を使っておらず，また，継続オブジェクトへの参照が`call/cc`からエスケープすることもなかったため，すべてのエスケープバリアを除去できた．

6.5 まとめ

この章では，第4章で示した遅延スタックコピー法の実装で必要となるエスケープバリアのオーバヘッドを軽減する手法として，プログラム変換により必要のないエスケープバリアを除去する手法を示した．まず，6.3節でScheme言語のサブセットであるScheme0言語を変換するアルゴリズムを示し，6.4.1節でフルセットのScheme言語に適用する拡張法を示した．これにより拡張したアルゴリズムでいくつかのベンチマークプログラムを変換したところ，大きな効果の見られるプログラムと効果の薄いプログラムがあった．そこで，アルゴリズムを拡張し関数の間で値の安全性に関する情報を受け渡す拡張を示した．これを用いると，6.3節のアルゴリズムだけでは効果の薄かったプログラムにも大きな効果が見られた．以上のように，6.3節で示したアルゴリズムを核としていくつかの改良をするとで，多くの不要なエスケープバリアを除去できることが分かった．したがって，最適化されたコードを生成するコンパイラなどにとって，この変換は十分に採用する価値があると言える．

第7章　おわりに

一級継続を持つプログラミング言語では，様々な制御の流れをユーザレベルで表現することができ，例外処理機構など特定の制御のパターンを表現する制御機能のみを持つプログラミング言語に比べて高い記述性を持つ．この特徴は，プログラマが一級言語を持つプログラミング言語を直接記述する場合に有利であるだけでなく，そのプログラミング言語が中間言語として使われる場合にも有利である．しかし，プログラミング言語処理系に一級継続を効率良く扱う機能を実装することは簡単ではない．また，他の言語機能と一級継続を組み合わせると，さらに記述性は高くなるが，処理系の実装は難しくなる．本研究では，このような，他の言語機能と一級継続の両方を持つ処理系の実装として，他言語呼出しとマルチスレッドを持つ処理系への一級継続の実装手法を提案した．さらに，関数フレームをスタックに生成する一般的な処理系に効率のよい一級継続を実装する手法も提案した．

他言語呼出しとマルチスレッドを持つ処理系への一級継続の実装は，本論文の第3章で扱った．ここでは，Java言語との相互呼出し機能を持ち，Java言語のマルチスレッドを扱うことができるScheme言語の処理系「ぶぶ」を対象に，一級継続を実装した．ぶぶでは，継続の実行には，その継続がキャプチャされた時にアクティブであった関数フレームがすべて必要になるのに対し，ぶぶではScheme言語の関数の関数フレームしかコピーすることができない．そこで，Java言語のメソッドからまだリターンしていない場合に限って，完全な継続として呼び出せる一級継続を実装した．この継続オブジェクトは，Java言語のメソッドからリターンした後は，Scheme言語の関数の実行だけを表す部分継続として振る舞う．このように実装した一級継続でも，多くの場面では完全な一級継続と同様に動作する．また，一級継続を使って記述されている既存のScheme言語のライブラリを使うことができるなど，有効に利用できる場面が多数あることを確認した．ここで用いた実装手法は，ぶぶのみに留まらず，高級言語からその処理系を記述している言語の関数を呼び出す機能を持ったプログラミング言語に一級継続を実装する際にも応用できる．

関数フレームをスタックに生成する一般的な処理系に，効率のよい一級継続を実装する手法としては，遅延スタックコピー法および，その改良手法を提案した．

第4章で示した遅延スタックコピー法では，継続がキャプチャされても，直ちには関数フレームのコピーを生成しない．代わりに，スタック上の関数フレームへのポインタを持つ継続オブジェクトを生成する．その上で，もし継続をキャプチャした関数からリターンした後もその継続が呼び出される可能性が残っていれば，継続をキャプチャした関数からのリターンの際に関数フレームをコピーする．さらに，第5章でスタックコピー共有法を示した．スタックコピー共有法は，スタック戦略と遅延スタックコピー法を使った処理系で，複数の継続オブジェクトの間でコピーした関数フレームを共有する手法である．従来のスタック戦略では，関数フレームのコピーを複数の継続オブジェクトの間で共有することが難しかったが，遅延スタックコピー法を導入することにより，容易に共有できるようになった．これらの手法により，関数フレームをスタックに生成する処理系で，効率の良い一級継続が実現できることを，実際にScheme処理系に実装して性能を評価することで示した．

また，遅延スタックコピー法のオーバヘッドであるエスケープバリアのうち一部を除去する手法を第6章で示した．遅延スタックコピー法では，継続オブジェクトへの参照が，その継続オブジェクトを生成した関数からりターンする前に，ヒープや大域変数に書き込まれる際に，継続オブジェクトにマークを付ける必要があった．第6章で示した手法は，実行前にプログラムのデータフローを解析し，マークを付ける必要がない値が書き込まれると分かっている書き込みのバリアを除去する．これにより，多くの不要なエスケープバリアが除去でき，最適化されたコンパイラに遅延スタックコピー法を採用する場合に問題となると予想されるオーバヘッドが克服できた．

以上の遅延スタックコピーとそれに関連する手法は，スタックに関数フレームを生成する処理系に一級継続を実装する際に有効な手法であると結論づけることができる．

参考文献

[1] Gosling, J., Joy, B., Steele, G. and Bracha, G.: *The Java Language Specification Second Edition*, Addison-Wesley.

[2] IEEE-1178-1990: *IEEE Standard for the Scheme Programming Language*, IEEE (1991).

[3] Kelsey, R., Clinger, W. and Rees, J.(eds.): *Revised^5 Report on the Algorithmic Language Scheme*, Vol. 11, No. 1, Kluwer Academic Publishers (1998).

[4] Goldberg, A. and Robson, D.: *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley (1983).

[5] Appel, A. W. and MacQueen, D. B.: Standard ML of New Jersey, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming* (Wirsing, M.(ed.)), New York, Springer-Verlag, pp. 1–13 (1991).

[6] Standard ECMA-335: *Common Language Infrastructure (CLI) Partitions I to V*, ECMA (2002).

[7] Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley.

[8] 山本晃成, 湯淺太一: 末尾再帰の最適化と一級継続を実現する為の JVM の機能拡張, 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG 11（PRO 12）, pp. 37–51 (2001).

[9] Clinger, W. D., Hartheimer, A. H. and Ost., E. M.: Implementation Strategies for First-Class Continuations, *Higher-Order and Symbolic Computation*, Vol. 12, No. 1, Kluwer Academic Publishers, pp. 7–45 (1999).

[10] Clinger, W. D. and Hansen, L. T.: Lambda, the ultimate label or a simple optimizing compiler for Scheme, *Proceedings 1994 ACM Conference on Lisp and Functional Programming*, ACM Press, pp. 128–139 (1994).

[11] Kelsey, R. A. and Rees, J. A.: A Tractable Scheme Implementation, *Lisp and Symbolic Computation*, Vol. 7, No. 4, pp. 315–335 (1994).

[12] 小宮常康, 湯淺太一: Future ベースの並列 Scheme における継続の拡張, 情報処理学会論文誌, Vol. 35, No. 11, pp. 2382–2391 (1994).

[13] Danvy, O.: Memory allocation and higher-order functions, *Interpreters and Interpretive Techniques (SIGPLAN'87)* (1987).

[14] Jaffer, A.: *SCM Manual* (2003).

[15] Flatt, M.: *PLT MzScheme: Langurage manual* (2000).

[16] Samples, A. D., Ungar, D. and Hilfinger, P.: SOAR: Smalltalk Without Bytecodes, *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, Vol. 21, pp. 107–118 (1986).

[17] Yuasa, T.: An Object-oriented Scheme System Bubu with Seamless Interface to Java, *Parallel and Distributed Computing for Symbolic and Irregular Applications* (Ito, T. and Yuasa, T.(eds.)), World Scientific, pp. 101–121 (2000).

[18] 窪田貴志, 湯淺太一, 倉林則之, 八杉昌宏, 小宮常康: Java 上の Scheme 処理系「ぶぶ」における単一のクラスローダを用いたオブジェクトシステムの実装, 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG 7（PRO 11）, pp. 57–69 (2001).

[19] 鵜川始陽, 湯淺太一, 小宮常康, 八杉昌宏: Java と相互呼び出し可能な Scheme 処理系「ぶぶ」における継続機能と例外処理機能の実装, 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG 11（PRO 12）, pp. 25–36 (2001).

[20] Bothner, P.: Kawa: Compiling Scheme to Java, Lisp Users Conference (1998).

[21] Anderson, K., Hickey, T. and Norvig, P.: Cilk: A Playful Combination of Scheme and Java, Rice University, CS Dept (2000).

[22] 鵜川始陽, 小宮常康, 八杉昌宏, 湯淺太一: スタックのコピーを遅延させる継続の実装方式, 日本ソフトウェア科学会第 19 回大会講演論文集, 日本ソフトウェア科学会 (2002).

[23] 鵜川始陽, 皆川宜久, 小宮常康, 八杉昌宏, 湯淺太一: 継続の生成におけるスタックコピーの遅延, 情報処理学会論文誌：プログラミング, Vol. 44, No. SIG 13（PRO 18）, pp. 72–83 (2003).

[24] Ugawa, T., Minagawa, N., Komiya, T., Yasugi, M. and Yuasa, T.: Lazy Stack Copying and Stack Copy Sharing for the Efficient Implementation of Continuations, *Proceedings of The First ASIAN Symposium on Programming Languages and Systems* (Ohori, A.(ed.)), Springer-Verlag, pp. 410–426 (2003).

[25] Yuasa, T., Nakagawa, Y., Komiya, T. and Yasugi, M.: Return Barrier, *Proceedings International Lisp Conference, San Francisco* (2002).

[26] Jones, R. and Lins, R.: *Garbage Collection*, JOHN WILEY & SONS (1996).

[27] 湯淺 太一他: TUTScheme マニュアル, http://www.yuasa.kuis.kyoto-u.ac.jp/~komiya/tus-man/tus/ (1998).

[28] Gabriel, R.: *Performance and Evaluation of Lisp Systems*, MIT Press (1985).

[29] Haynes, C. T., Friedman, D. P. and Wand, M.: Continuations and Coroutines, *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pp. 293–298 (1984).

[30] 清野智弘, 伊藤貴康: PaiLisp の並列構文の実現法と評価, 情報処理学会論文誌, Vol. 34, No. 12, pp. 2578–2591 (1993).

[31] Baker, H. G.: CONS Should not CONS its Arguments, or, a Lazy Alloc is a Smart Alloc, *ACM SIGPLAN Notices*, Vol. 27, No. 3, pp. 24–34 (1992).

[32] Friedman, D. P. and Wise, D. S.: The One-Bit Reference Count, *BIT*, Vol. 17, No. 3, pp. 351–359 (1977).

[33] Mohr, E., Kranz, D. A. and Halstead, Jr., R. H.: Lazy task creation: a technique for increasing the granularity of parallel programs, *Proceedings of the 1990 ACM conference on LISP and functional programming*, ACM Press, pp. 185–197 (1990).

[34] Goldstein, S. C., Schauser, K. E. and Culler, D. E.: Enabling Primitives for Compiling Parallel Languages, *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers* (1995).

[35] Taura, K. and Yonezawa, A.: Fine-grain Multithreading with Minimal Compiler Support - A Cost Effective Approach to Implementing Efficient Multithreading Languages, *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 320–333 (1997).

[36] Chakravarty, M. M. T.: Lazy Thread and Task Creation in Parallel Graph Reduction, *Lecture Notes in Computer Science*, Vol. 1467, pp. 231+ (1998).

[37] Choi, J.-D., Gupta, M., Serrano, M. J., Sreedhar, V. C. and Midkiff, S. P.: Escape Analysis for Java, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 1–19 (1999).

[38] 小川基弘, 小松秀昭, 中谷登志男: Java言語に対する効果的なNullチェックの最適化手法, 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG 2（PRO 9）, pp. 81–96 (2001).

謝辞

本研究を行うにあたり丁寧な御指導を賜わった，京都大学教授の湯淺太一先生に厚くお礼申し上げます．

湯淺先生にはこの研究を行う機会を与えて頂き，また，研究を進めるに当たってご指導，ご助言，ご協力頂きました．京都大学助教授の八杉昌宏先生，当時京都大学助手で現在豊橋技術科学大学講師の小宮常康先生にも貴重なご助言を頂きました．厚くお礼申し上げます．

当時湯淺研究室でぶぶオブジェクトシステムの開発をなさっていた窪田貴志氏には，ぶぶについての研究でご助力頂きました．湯淺研究室の皆川宜久氏，米田匡史氏には，遅延スタックコピー法の研究に協力して頂きました．特に皆川宜久氏には，遅延スタックコピー法を実装したMzScheme処理系を提供して頂きました．

最後に，貴重なご意見を頂きました京都大学教授の富田眞治先生，奥乃博先生をはじめとする情報学研究科の皆様に深く感謝致します．

研究業績リスト

[1] 鵜川始陽, 湯淺太一, 小宮常康, 八杉昌宏: Java と相互呼び出し可能な Scheme 処理系「ぶぶ」における継続機能と例外処理機能の実装, 情報処理学会論文誌：プログラミング, Vol. 42, No. SIG 11（PRO 12）, pp. 25–36 (2001).

[2] 鵜川始陽, 小宮常康, 八杉昌宏, 湯淺太一: スタックのコピーを遅延させる継続の実装方式, 日本ソフトウェア科学会第 19 回大会講演論文集 (2002).

[3] 鵜川始陽, 皆川宜久, 小宮常康, 八杉昌宏, 湯淺太一: 継続の生成におけるスタックコピーの遅延, 情報処理学会論文誌：プログラミング, Vol. 44, No. SIG 13（PRO 18）, pp. 72–83 (2003).

[4] Ugawa, T., Minagawa, N., Komiya, T., Yasugi, M. and Yuasa, T.: Lazy Stack Copying and Stack Copy Sharing for the Efficient Implementation of Continuations, *Proceedings of The First ASIAN Symposium on Programming Languages and Systems* (Ohori, A.(ed.)), Springer-Verlag, pp. 410–426 (2003).

[5] Ugawa, T., Yoneda, M., Yasugi, M. and Yuasa, T.: Removing Unnecessary Escape Barriers for Continuations (2004). poster session.